1996

A bipartite model of distributed systems: Possibilities and implications

Anna Karin Brunstrom
College of William & Mary - Arts & Sciences

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A BIPARTITE MODEL OF DISTRIBUTED SYSTEMS: POSSIBILITIES AND IMPLICATIONS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Anna Brunstrom

1996

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Anna Brunstrom

Approved, June 1996

Phil Kearns
Thesis Advisor

Rahul Simha

Weizhen Mao

Gianfranco Ciardo

Stephen Clement
Department of Geology

Contents

ACKNOWLEDGMENTS

The author wishes to thank her advisor, Dr. Phil Kearns, for his excellent guidance throughout the development of this dissertation, and for his inspiring enthusiasm for the field of Computer Science. The author also wishes to thank Dr. Rahul Simha for his careful reading of an earlier draft of this work and for many useful discussions on the material contained in this dissertation. Finally, the author wishes to thank her fellow graduate students, with a special thanks to Felipe Perrone and Jean Mayo, for many helpful discussions and for good company over the years when this work was carried out.

List of Tables

List of Figures

ABSTRACT

Networking software is generally designed in layers. User processes exist at the application layer. They rely on the transport layer to provide them with end-to-end communication. In the distributed systems literature communication is traditionally viewed from the application layer. At the application layer we have no knowledge of the whereabouts of a message once a send operation is completed. At the transport layer, on the other hand, information about the delivery of a message to the transport layer in the receiving host is often available. We believe transport layer information can be better utilized in distributed systems design. This dissertation presents support for this thesis.

We first develop a bipartite system model that allows us to reason formally about transport layer information. We can then propagate transport layer information to the application layer in a formally sound fashion. We define two constructs, delivered and delivered_all, for this purpose. The constructs allow easy implementation of message ordering protocols at the user level. We develop user-level implementations of both causally ordered communication and flush channels.

We also consider the impact of transport layer information on vector time. Transport layer vector time can improve both the computational efficiency and the accuracy of results for certain algorithms. Transport layer vector time also provides the possibility of updating vector time for acknowledgment messages. A distributed termination detection algorithm that takes advantage of this possibility is designed.

Finally, we provide a prototype implementation and associated experimental results. We design and implement a transport layer protocol with support for the delivered and delivered_all constructs. Our flush channel implementation is also included in the prototype system. Our experimental results verify the feasibility of our implementation and show practical evidence in support of the usefulness of transport layer information.

A BIPARTITE MODEL OF DISTRIBUTED SYSTEMS: POSSIBILITIES AND IMPLICATIONS

Chapter 1

Introduction and Motivation

This chapter introduces our thesis. A discussion on how this thesis relates to other current trends in the research community is provided. Finally, a more detailed overview of the remainder of this dissertation is presented.

1.1 Thesis

Networking software is traditionally developed in layers, where each layer is responsible for a particular aspect of communication. The TCP/IP protocol suite, for example, is usually regarded as a four-layer system[98] whereas the ISO reference model provides a seven-layer view of the communication software[100]. In the TCP/IP protocol suite, as well as in most models, the layer responsible for providing end-to-end communication is the *transport layer*. A transport layer protocol at a source node accepts messages from an application and forwards them to the corresponding transport layer protocol on the destination host for delivery to the destination application. While communicating with the transport layer in another host the transport layer protocols rely on services provided by

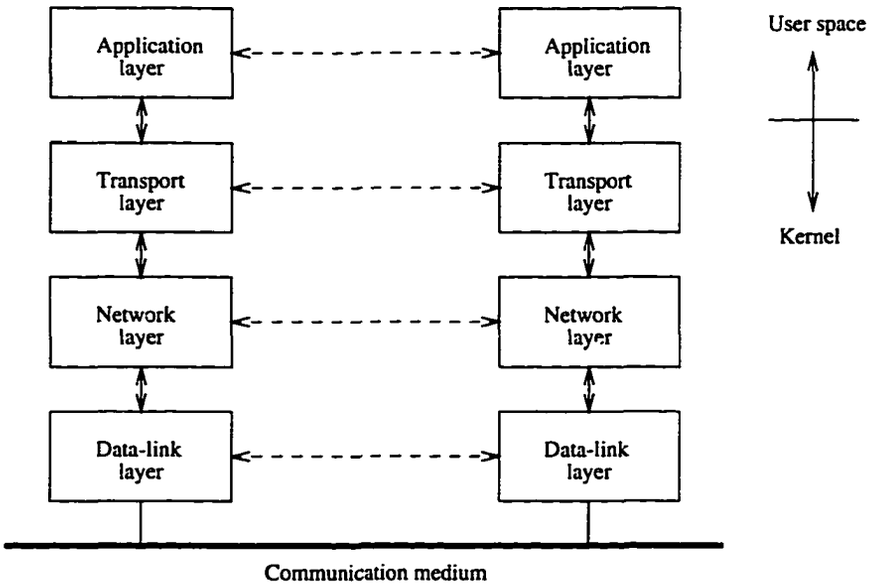

Figure 1.1: TCP/IP Communication Model

the layers underneath it, the network layer and the datalink layer in the TCP/IP protocol suite. The services provided by transport layer protocols vary from reliable fully-ordered connection-oriented service, as provided by TCP[80], to unordered and unreliable message delivery, as provided by UDP[77]. In most systems the transport layer protocols and the protocols of lower layers are implemented in the kernel. The applications utilizing the services of the transport layer belong to the application layer. Application layer processes include general user processes, communicating through (for example) BSD-style sockets, as well as well-known application services provided by most systems, such as FTP and Telnet. All application layer processes exist in user space. The layered view of communication used by the TCP/IP protocol suite is depicted in Figure 1.1.

Most algorithms developed to solve problems in distributed systems are designed at the application level. Their design is based on communication as perceived at the application

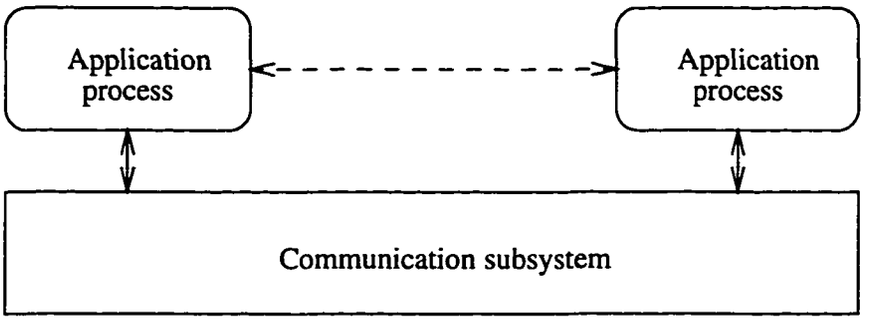

Figure 1.2: System View Taken by Distributed Algorithms

layer. The operations of the lower layers are only implicitly defined by the system assumptions made by the algorithms. A distributed system, P, is generally modeled as a set of N processes, $P = \{P_0, P_1, \ldots, P_{N-1}\}$, which communicate only through message passing; there is no shared memory or global clock in a distributed system. The message passing system is traditionally assumed to provide reliable asynchronous communication. There is no known upper bound on the delay of messages but messages are assumed to eventually reach their destinations. In addition, some algorithms may require the underlying channels to be FIFO. The system as viewed by most algorithms in distributed systems is illustrated in Figure 1.2.

Asynchronous message passing as viewed from the application layer operates in a "fire-and-forget" fashion. Once a message is sent, the user has no knowledge of when it will reach its destination. A process executing a *send* operation returns immediately. The message is copied by the networking support code which is responsible for delivering the message to the receiver. Assuming reliable message passing, the transport layer in the sender must maintain information on the status of messages. To ensure reliable end-to-end communication the transport layer must keep the copy of the message in its buffers until it

knows that it has been correctly delivered to the transport layer of the receiver. Thus, even though no information on the whereabouts of messages is available at the user level, this information is available at the kernel level. In fact, information about delivery of messages is available virtually for free at the transport layer.

Our thesis is that low-level information, available at the transport layer, may be used to our advantage when solving problems in distributed systems. As will be explored in this dissertation, knowledge of the delivery of messages can be a useful tool for distributed systems design.

Both formal and practical aspects of the use of low-level information will be considered. We will develop a bipartite system model that models both the application layer and the transport layer in the receiving host. Our bipartite system model will allow us to reason formally about transport layer information. Based on our model, we can develop methods to propagate transport layer information to the user level. We will show how this information can be applied to design alternative solutions to several well-known problems in distributed systems. Finally, we will describe a prototype implementation and experimental results in order to show that our ideas are not only theoretically sound and academically appealing, but also implementable in practice.

1.2 The Thesis in a Bigger Context

In this section we examine how our thesis relates to some other main ideas which inspire today's researchers. We will merely attempt to illustrate how our thesis fits in the context of some current research trends. Additional related work will be presented throughout our development as we tackle each aspect of the utilization of transport layer information.

Better utilization of transport layer information and propagation of transport layer

information to the application layer are consistent with the *end-to-end* argument[88]. According to the end-to-end argument very few functions should be placed at a low level in the implementation of a distributed system. The power should be available at a high level since this is where the functionality must ultimately be implemented. Low-level functions are not sufficient at a higher level and only serve as performance enhancers. If we want functionality in our system to be implemented at the application layer, then we cannot hide power from this level. To reveal as much power as possible is one of the cardinal rules in operating system design[54].

Inspired in part by the end-to-end argument, many recent research efforts have considered how to implement network protocols at the user level[65, 102, 56, 67]. Ease of extensibility, support for multiple protocols providing different services, and the ability to utilize application-specific knowledge to improve performance are identified as the main advantages of a user-level implementation[102]. As we will see, some of the applications considered in this dissertation will be user-level implementations of network protocols. Information propagated from the transport layer to the application layer can be utilized to design flexible and straightforward user-level implementations of message ordering protocols.

The effort to implement networking protocols at the user level is illustrative of the general trend in operating system design today. Many current operating systems are large, inflexible and poorly matched to the quickly changing needs of the applications[11, 24, 33]. A number of current research projects focus on providing a small operating system kernel which provides a core of basic services and the tools for flexible extensions to the kernel or for implementing additional services at the user level. Well-known micro kernels include Mach[2] and Amoeba[66]. Recent work on extensible operating system kernels include

SPIN[11], the cache kernel[24], and the exokernel[33].

As described earlier, network software is commonly developed in layers. Although this makes for a clear division of labour and a logical design, it can also lead to poor implementations of network code. Redundancy between the layers can account for a major portion of the execution cost. Early implementations of network code often copied the data to be transmitted between buffers at each layer, adding an substantial overhead to the implementation. With the emergence of high-speed networks, the execution cost of communication software has become predominantly more important. Today, multiple layers are being squeezed for efficiency and unnecessary buffer copying is being removed[72, 1, 28]. Presenting transport layer information to the user level can also reduce redundancy. It is often the case that an algorithm sends explicit acknowledgment messages at the user level. Allowing information about transport layer acknowledgments to be propagated to the user level may alleviate the need for user-level acknowledgments. One of the algorithms which will be developed in our dissertation is a distributed termination detection algorithm. Our algorithm utilizes information provided by acknowledgments at the transport layer. The use of low-level information makes our algorithm a simple and efficient solution to the distributed termination detection problem. Our algorithm would not be feasible if user-level acknowledgments had been used due to the high message passing overhead such acknowledgments would introduce.

As this section has illustrated, the thesis promoted in this dissertation fits in well with many of today's research trends. The appropriate use of transport layer information can aid in the design of flexible user-level protocols as well as reduce redundancy.

1.3 Overview of the Dissertation

This section gives an overview of the material presented in the reminder of this dissertation. To reason formally about low-level information, we first develop a bipartite system model which allows us to make a distinction between information available at the application layer and at the transport layer. In particular, our system model must allow us to distinguish between when a message is *delivered* at the transport layer and when it is *received* at the application layer.[1] The model currently used for reasoning about asynchronous message passing systems was developed by Schlichting and Schneider[90]. It does not allow such a distinction. Schlichting and Schneider's model considers communication between processes at the application layer. All lower layers are combined into a single entity which represents the network. Our system model will be a straightforward extension of Schlichting and Schneider's model. Our bipartite system model explicitly models the transport layer in the receiver as well as the application layer processes. Based on our system model we can then formally define two constructs, *delivered* and *delivered_all*, which will allow us to propagate low-level information to the user level in a sound fashion. Our bipartite system model as well as our *delivered* and *delivered_all* constructs will be developed in Chapter 2.

In the next few chapters we illustrate how our constructs, and the transport layer information they provide, can be used to solve some sample problems in distributed systems. As mentioned earlier, one major application of our constructs is in the design of message ordering protocols at the user level. As long as messages are passed in FIFO order from the transport layer to the application layer, knowledge of delivery of messages is as powerful as knowledge of receipt of messages in terms of order. As a result, our constructs

[1] Our use of *delivered* and *received* is based on the work by Schlichting and Schneider[90]. We warn the reader that some authors use the words with the opposite meaning.

are extremely useful for providing application-specific ordering constraints. We will present user-level implementations of both causally consistent message passing[89] in Chapter 4 and flush channels[5] in Chapter 3. As an extension of causally consistent message passing we will also define causal consistency and causally early delivery for an individual message in Chapter 3. For some applications causally consistent message passing enforces more ordering constraints than necessary. Allowing ordering constraints to apply to specific messages can greatly increase the efficiency of such algorithms.

As mentioned above no common clock exists in a distributed system. Events in the system are only partially ordered based on causality[53]. Vector time was introduced, by Mattern[57] and Fidge[34] independently, as a means for capturing the causal relationship in the system. It has become a standard tool used in the design of algorithms for distributed systems. Vector time is updated through communication[2]. As we consider the impact of transport layer information, it is therefore important to consider vector time as perceived at the transport layer. This is the topic of Chapter 5. We establish formal relationships between application layer and transport layer vector time and show how transport layer vector time can improve performance, both in terms of quality of results and in terms of computational efficiency, for certain types of algorithms. Transport layer vector time can also allow us to update vector time for acknowledgment messages, a possibility which we also explore in Chapter 5.

Chapter 6 discusses our prototype implementation. We have implemented a transport layer protocol for reliable unordered message passing, with support for the *delivered* and *delivered_all* system calls, within the realm of the Linux operating system. Our user-level

[2]Communication events always update vector time. It is also possible to allow local events to update vector time. The formal definition of vector time is given in Chapter 5.

routines for flush channel communication have also been implemented. Experimental results from our prototype implementation are discussed in Chapter 7. Finally, Chapter 8 contains concluding remarks and directions for further research.

Chapter 2

Formal Aspects

In this chapter we develop the formal tools needed in order to reason about transport layer information. We begin by developing a bipartite system model which lets us distinguish between when a message is delivered and when a message is received. We then proceed to define two user-callable constructs, *delivered* and *delivered_all*, which convey transport layer information to the user layer. The proof rules for our system are established and their use is illustrated through an example.

2.1 System Model

As mentioned in the introduction, the traditional model used for reasoning about asynchronous message passing was developed by Schlichting and Schneider[90]. The model developed by Schlichting and Schneider makes no distinction between when a message is *delivered* and when a message is *received*. Ignoring the delivery of a message to the transport layer of the receiver is justified if we reason strictly about receipt of messages. However, we are interested in using transport layer information on when a message is delivered. The

transport layer support code at the sender gives up responsibility for the message when it has been delivered to the transport layer in the receiver. This could happen long before the message is received by the application program in the receiving process. When needed, the transport layer will buffer an incoming message until a *receive* system call is performed by the application. In a system employing reliable message passing, information about delivery of messages is available to the kernel as a result of the natural operation of the transport layer. In contrast, knowledge of when a message is received cannot be obtained without extra message passing overhead. Thus, a distinction between the two operations is necessary. As our review of Schlichting and Schneider's model will show, it is not powerful enough to allow reasoning about delivery of messages. In this section we will therefore develop a bipartite system model which allows us to reason formally about information available at the transport layer. Our model is easily derived from Schlichting and Schneider's traditional model. The proof obligations for the communication statements in our system can be directly adopted from the rules derived by Schlichting and Schneider. Since both models use CSP processes to model communication we will begin the section with a brief introduction to CSP[40].

2.1.1 Introduction to CSP

CSP was introduced by Hoare[40] as a mechanism for programming parallel processes. There are four simple commands in CSP: assignment (:=), *skip* (the null command), and two communication commands corresponding to send and receive. Communication in CSP is synchronous. Communication takes place between two processes P_0 and P_1 when P_0 specifies P_1 as the destination of its output and P_1 specifies P_0 as the source of its input or vice versa. When the communicating processes have both reached their respective commu-

nication commands the commands are *ready* and the information exchange can take place. Either of the two processes might be *blocked* waiting for the other process to reach its input or output command. A communication command *fails* when the process with which it tries to communicate is terminated. An output command has the form:

$$P_r\,!\,expr$$

where P_r indicates the process which is the destination of the output, and *expr* is the expression whose value is the data to be output. In terms of message passing, P_r is the receiver of the message *expr*. An input command has the form:

$$P_s\,?\,var$$

where P_s indicates the process which is the source of the input command, and *var* is the target variable for the input data. In terms of message passing, P_s is the sender of the message received in *var*.

Additional commands are formed by composition of the simple commands. The composite commands we need are *sequence* and *repetition*. Just as one would expect, a sequence command, $S_1; \ldots ; S_n$, executes S_1 through S_n in sequence. A repetitive command has the form:

$$\textbf{do} < guarded\,command > \{[] < guarded\,command >\}\,\textbf{od}$$

where a guarded command has the form:

$$< guard > \longrightarrow < command\,list >$$

A guarded command is executed only when its guard is ready. A guard consists of a possibly empty sequence of boolean expressions possibly followed by a communication statement. The guard is evaluated and executed from left to right. A guard is ready for execution if all the boolean expressions evaluate to true and the communication statement is ready. The guard is blocked if the boolean expressions are true and the communication statement is blocked. If a boolean expression evaluates to false or the communication statement fails, then the guard fails. A repetitive command repeatedly selects, at random, a command for execution among the guarded commands whose guards are ready. When all guarded commands fail the repetitive command exits.

2.1.2 Traditional System Model

The system model for asynchronous communication, developed by Schlichting and Schneider, consists of three processes. The sending and receiving processes communicate through a third process which represents the network. Processes in the model communicate using CSP-style communication. Modeling communication using CSP processes allows proof rules derived from the model to use a proof methodology for CSP which is known to be both sound and relatively complete[55]. The system model established by Schlichting and Schneider is depicted in Figure 2.1. As we can see, the model conforms to the view of the system used by most algorithms in distributed systems, illustrated in Figure 1.2. All layers below the application layer as well as the communication medium are combined into one process representing the network. The state of the network is represented by implicit variables. For unordered message passing, a *send* operation inserts the message in a send multiset, σ, and a *receive* operation inserts the message in a receive multiset, ρ. The received message must be a member of $\sigma \ominus \rho$ prior to the execution of the receive statement, where $\ominus$ represents

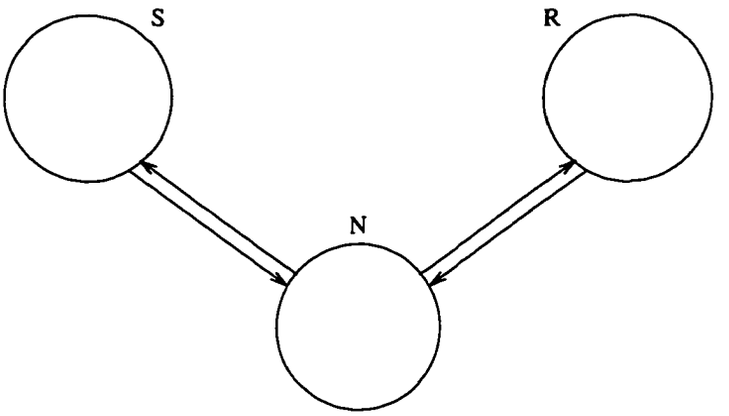

Figure 2.1: Traditional System Model

the multiset difference operator. The messages in $\sigma \ominus \rho$ are messages which have been sent but not yet received. If FIFO communication is used, then σ and ρ are sequences rather than multisets and the received message is the head of $\sigma - \rho$, where $\sigma - \rho$ is the sequence difference[1]. The receiver must receive the earliest message sent among all outstanding messages. The model is most easily understood by examining an example. Let us consider FIFO communication. Simulating a virtual circuit using the model is straightforward. The sending process, S, and the receiving process, R, communicate through the network process N:

$N :: \sigma, \rho := \emptyset, \emptyset;$

do

$S?\sigma \longrightarrow$ skip

$[](\sigma - \rho) \neq \emptyset;\ R!(hd(\sigma - \rho), \rho + hd(\sigma - \rho)) \longrightarrow$ skip

od

[1]The sequence difference, $s_1 - s_2$, is obtained by deleting prefix s_2 from the beginning of s_1. It is undefined if s_2 is not a prefix of s_1.

The symbol $\emptyset$ is used to denote an empty sequence, the $+$ operator appends an element to a sequence, and $hd(\cdot)$ returns a copy of the element at the head of a sequence. An application-level asynchronous send statement *send*(*expr*) is modeled by:

$$N\,!\,\sigma + expr$$

and a receive statement *receive*(m) is modeled by:

$$N\,?\,(m, \rho)$$

We can see that the network process executes a loop, a repetitive command in CSP, receiving messages from the sender or sending messages to the receiver. All the work performed by the network process is carried out in the guards of the repetitive command. The network process terminates when the sending process has terminated and there are no outstanding messages or when both the sending and the receiving processes have terminated. As seen above the sending of a message is modeled by inserting the message into σ. When the sending process and the network process communicate it is as if the distributed assignment $\sigma := \sigma + expr$ had been carried out. When a message is received it is inserted into ρ as well as passed to the receiver. Communication between the network process and the receiving process corresponds to the distributed assignments $m := hd(\sigma - \rho)$ and $\rho := \rho + hd(\sigma - \rho)$. As ensured by the boolean expression in the second guard, communication between the network process and the receiving process can only occur when $\sigma - \rho$ is nonempty.

2.1.3 Bipartite System Model

In the model developed by Schlichting and Schneider there is no distinction between when a message is *delivered* and when a message is *received.* No such distinction is possible since all lower layers are represented by a single process. In order to construct a model which

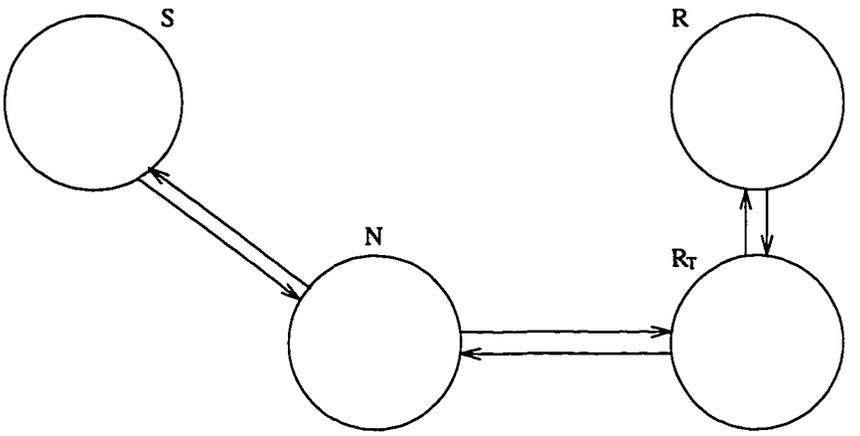

Figure 2.2: Extended System Model

lets us distinguish between the delivery of a message at the transport layer and the receipt of the message at the application layer, we will extend Schlichting and Schneider's model with an additional process, which represents the transport layer at the receiver. The state of the transport layer in the receiver can be represented by adding an additional implicit variable, ρ_T. When a message is delivered it is added to ρ_T and when it is received it is added to ρ. Our extended system model is illustrated in Figure 2.2.

As mentioned earlier, in this dissertation we focus our attention on a communication system which provides reliable unordered message passing. The transport layer at the receiver is assumed to pass messages to the application in the same order they were delivered. Communication between a sender S and a receiver R can be simulated by our CSP model as follows. A send statement in S, *send*(*expr*), is simulated by:

$$N\,!\,\sigma \oplus expr$$

where the $\oplus$ operator adds an element to a multiset. The sending process S communicates with the network process N:

$N :: \sigma, \rho_T := \emptyset, \emptyset;$

do

$S\,?\,\sigma \longrightarrow$ skip

$[]\,(\sigma \ominus ms(\rho_T)) \neq \emptyset;\ R_T\,!\,(\rho_T + ch(\sigma \ominus ms(\rho_T)) \longrightarrow$ skip

od

$Ch(\cdot)$ chooses one of the elements from a multiset and $ms(\cdot)$ converts a sequence to the corresponding multiset. The $ch(\cdot)$ function is used since we are modeling unordered communication. The $ms(\cdot)$ operation is necessary since ρ_T, as well as ρ, must be modeled as a sequence to allow us to pass messages to the application in FIFO order. The transport layer process at the receiver, R_T, now acts as the second communication partner for the network process:

$R_T :: \rho, \rho_T := \emptyset, \emptyset;$

do

$N\,?\,\rho_T \longrightarrow$ skip

$[]\,(\rho_T - \rho) \neq \emptyset;\ R\,!\,(hd(\rho_T - \rho), \rho + hd(\rho_T - \rho)) \longrightarrow$ skip

od

The application layer process in the receiver, R, communicates with the receiving transport layer process where a receive statement, *receive*(m), is simulated by:

$R_T\,?\,(m, \rho)$

We can see that modeling communication using our extended system model is very similar to communication modeling in the traditional system model. The transport layer process at the receiver, R_T, simply works as an additional buffer process with behavior similar to the behavior of the network process.

2.1.4 Proof Obligations

Having used our model to simulate communication in our system, we next need to establish the appropriate proof rules. As described by Levin and Gries[55] a partial correctness proof[2] of a distributed program consists of three parts. The first step is to give a *proof in isolation*, also called a *sequential proof*, for each process that is part of the program. This is done by giving a consistent Hoare-style [39] annotation of each process. For an executable statement, S, a Hoare triple $\{P\}S\{Q\}$ means that if statement S is executed from a state satisfying P, then Q will be true at the termination of S. P is the precondition of S, $pre(S)$, and Q is the postcondition of S, $post(S)$. Some communication statements are miraculous in isolation. If a statement never terminates in isolation, then anything could be asserted as the postcondition of that statement. The validity of such miraculous assertions is ensured by the second step, the *satisfaction proof*. Finally we need to ensure that statements in one process do not invalidate assertions made in another process. This is established through a *noninterference proof*.

The proof rules for unreliable datagrams were derived by Schlichting and Schneider[90]. Our system supports reliable unordered communication or, in other words, reliable datagrams. The proof rules derived for unreliable datagrams are still valid for reliable datagrams. Unreliability is modeled by leaving lost messages in $\sigma \ominus \rho$ forever. Unreliability does not show up in the developed proof rules. We can easily adapt Schlichting and Schneider's proof rules to obtain the proof rules for *send* and *receive* in our system. We do not allow a boolean condition as part of our *receive* statement. This slightly simplifies the receive axiom compared to Schlichting and Schneider's receive axiom. We also need to modify the

[2] A partial correctness proof ensures that the program behaves correctly if it makes progress (safety). It does not ensure freedom from deadlock (liveness).

proof rules to account for our addition of ρ_T. We derive the following axioms, used during the proof in isolation:

Network Axioms: $ms(\rho_T) \subseteq \sigma; \quad ms(\rho) \subseteq \sigma; \quad \rho \leq \rho_T$

Send Axiom: $\{W^{\sigma}_{\sigma \oplus expr}\}\ send(expr)\ \{W\}$

Receive Axiom: $\{R\}\ receive(m)\ \{Q\}$

where the notation P^y_x means that every free occurrence of y in P is textually replaced by x. The *receive* statement is miraculous in isolation. To establish satisfaction, we have the following proof obligation:

Satisfaction Proof: For every *receive* statement r, prove the validity of

$$Sat_{asynch}(r)\text{: } (pre(r) \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho)) \Rightarrow Q^{m,\rho}_{MTEXT,\rho+MTEXT}$$

Note that $((\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho)) \Rightarrow (MTEXT \in (\sigma \ominus ms(\rho)))$. To show noninterference for *send*s and *receive*s we must establish:

Noninterference Proof: For every *send* statement, s, and every assertion, I, parallel to s, show

$$NI_{asynch}(s, I) : \{pre(s) \wedge I\} s \{I\}$$

For every *receive* statement, r, and every assertion, I, parallel to r, show

$$NI\_Sat_{asynch}(r, I)\text{: } (pre(r) \wedge I \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho)) \Rightarrow$$

$$I^{m,\rho}_{MTEXT,\rho+MTEXT}$$

As expected, we can see that our extension of the system model to allow reasoning about delivery of messages does not really affect the proof obligations for *send* and *receive*

statements in the system. The proof rules are only strengthened to allow reasoning about transport layer information. Programs which do not employ transport layer information are not affected. The network axiom is strengthened to express the relationships between ρ_T and σ and ρ_T and ρ. The send and receive axioms remain as derived by Schlichting and Schneider. The antecedents for the satisfaction formula and for the second noninterference formula are slightly altered. However, as noted above, $((\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho)) \Rightarrow (MTEXT \in (\sigma \ominus ms(\rho)))$. Thus, any proof based on the weaker assumption, $MTEXT \in (\sigma \ominus \rho)$, used by Schlichting and Schneider is still valid under the proof rules derived above. The stronger assertions about the state of the transport layer are needed for programs which utilize transport layer information. This will become evident as we examine an example in Section 2.3.

2.2 The Delivered and Delivered_all Primitives

The previous section introduced our model of the system and established the proof obligations for *sends* and *receives* in our system. In this section we can now define two new primitives, *delivered* and *delivered_all*, which will allow us to utilize the low-level knowledge of when a message is delivered at the transport layer of the receiver in a formally sound way.

Our *delivered* primitive takes a message identifier as a parameter. A call to *delivered* will return when the message indicated by the message identifier has been delivered at the transport layer of the receiver. Thus, by definition of *delivered* we have:

$$delivered(id(m)) \Rightarrow m \in \rho_T$$

where $id(m)$ is the message identifier corresponding to message m. *Delivered* is a monotonic property which remains true forever once it is established. *Delivered_all* takes no parameters. When a call to *delivered_all* returns, all messages sent prior to invoking *delivered_all* have been delivered to the transport layer of the receiving process. By definition of *delivered_all* we have:

$$delivered\_all() \Rightarrow (ms(\rho_T) = \sigma)$$

Delivered_all is a nonmonotonic property. It remains true until another *send* operation is performed, and thus can only be invalidated by the sending process itself.

The proof rules for *delivered* and *delivered_all* are easily established. The primitives are miraculous in isolation[3] which gives us the following axioms:

Delivered Axiom: $\{S\}\, delivered(id(m))\, \{Q\}$

Delivered_all Axiom: $\{R\}\, delivered\_all()\, \{U\}$

Validity for the primitives is established by a satisfaction proof. Immediately before *delivered(id(m))* returns, the state can be described by $(S \wedge (m \in \rho_T))$, and immediately before *delivered_all* returns, the state can be characterized by $(R \wedge (ms(\rho_T) = \sigma))$[4]. Thus in order to establish satisfaction we have the following proof obligation:

Satisfaction Proof: For every $delivered(id(m))$ statement d, prove

$$Sat_{del} : (pre(d) \wedge (m \in \rho_T)) \Rightarrow Q$$

[3]The axioms are not sound in isolation since it is possible for a process to send messages to itself in an asynchronous message passing system. This is not a problem since the satisfaction proof will ensure validity. The receive axiom is unsound in isolation for the same reason[90].

[4]The characterization of the states assumes a correct noninterference proof.

and for every *delivered_all* statement a prove

$$Sat_{dall} : (pre(a) \wedge (ms(\rho_T) = \sigma)) \Rightarrow U$$

Noninterference follows trivially from the Delivered Axiom and the Delivered_all Axiom for *delivered* and *delivered_all* in isolation. We must also establish noninterference in accordance with satisfaction as given by:

$$NI\_Sat_{del}(d, I) : (I \wedge pre(d) \wedge (m \in \rho_T)) \Rightarrow I$$

$$NI\_Sat_{dall}(a, I) : (I \wedge pre(a) \wedge (ms(\rho_T) = \sigma)) \Rightarrow I$$

NI_Sat_{del} and NI_Sat_{dall} are also trivially true. Thus, we see that noninterference can be ignored for *delivered* and *delivered_all*. This could have been established by noting that *delivered* and *delivered_all* do not modify the state of any variables, as well as through explicitly expressing the proof obligations.

In a system that supports reliable message passing, the *delivered* and *delivered_all* primitives can be implemented by the kernel at very low cost. To ensure reliable delivery the networking support code has to buffer sent messages until they are known to be delivered. Thus, all the kernel has to do to implement the primitives is to check the transport layer send buffers. For example, when *delivered* is invoked, the buffers are checked for the indicated message. If the message is present in the buffers, then the caller will be suspended until the message is cleared from the buffers. The message identifier used by *delivered* would, in practice, refer to the sequence number or sequence number offset of the message. Reuse of sequence numbers by the transport layer is not a problem with respect to correct-

ness. An application that performs a *delivered* call with an old sequence number that has already been reused will not return until the new message has been delivered. This could cause an unnecessary wait, but the semantics of *delivered* are still preserved. The fact that the transport layer is again using the sequence number implies that the earlier message has been correctly delivered. Normally, an application is concerned about the delivery of recently sent messages and the reuse of sequence numbers is not an issue. Our prototype implementation of *delivered* and *delivered_all* will be discussed in Chapter 6.

Although the distinction between delivery and receipt of messages is sometimes made in the literature, for instance when considering implementations of transport layer protocols, we have not encountered any discussion formalizing the differences. To our knowledge, our *delivered* and *delivered_all* primitives do not have any clear counterparts in the literature. An attempt to provide low-level information to the user is made in Psync [73]. Psync is a system supporting causally ordered broadcasts, where causal order is implemented by maintaining a view of the message context graph at each processor. The user is provided with primitives for retrieving information from the context graph. The user can test whether a message m is *stable*, where m is considered stable once every other process has sent a message in the context of m. Thus, unless all processes send messages, m will never become stable. A primitive for determining if there are any *outstanding* messages is also provided. A message is defined to be outstanding if it is a member of the context graph but it is not yet in the process' view of the context graph. This is not a sound construct. It is impossible to determine whether a message is en route to a process. The property is also nonmonotonic and can be invalidated by statements outside the control of the process using the construct.

One of the most attractive features about the *delivered* and *delivered_all* primitives is that they are as powerful as *received* and *received_all* in terms of ordering, where *received*

and *received_all* have the obvious interpretation. This is true as long as the transport layer in the receiver passes messages to the application layer in an order consistent with the order messages were delivered at the transport layer. It is expressed by the following order conservation property:

Order Conservation Property: If the transport layer in the receiver passes messages in FIFO order with respect to delivery order, then

$$(T(delivered(id(m))) < T(delivered(id(m')))) \Rightarrow$$

$$(T(received(id(m))) < T(received(id(m'))))$$

where $T(delivered(id(m)))$ indicates the time at which message m was delivered. When unordered message passing is used, the normal procedure would be for the transport layer to pass messages to the application in FIFO order and the Ordering Conservation Property can be applied. In the next chapters we will see how our *delivered* and *delivered_all* primitives in combination with the Order Conservation Property can be used to implement ordering constraints on top of an unordered communication channel.

In this dissertation we focus on a system that uses unordered message passing. However, the semantics of *delivered* and *delivered_all* are independent of this fact. Only what can be deduced from the information given by the primitives and the usefulness of the primitives depend on the communication paradigm used. For a system communicating over FIFO channels, the Order Conservation Property described above does not usually hold. Most implementations of FIFO communication rely on the transport layer in the receiver to ensure the correct receipt order. The sender includes a sequence number in each message, from

which the correct receipt order can be deduced. The transport layer in the receiver passes messages to the application in the order they were sent, as indicated by their sequence numbers, rather than in the order they were delivered. Over a FIFO channel, the messages are fully ordered and we can establish a stronger order conservation property between sends and receives:

Strong Order Conservation Property: If FIFO communication is used, then

$$(T(send(m)) < T(send(m'))) \Rightarrow$$

$$(T(received(id(m))) < T(received(id(m'))))$$

The Strong Order Conservation Property expresses the semantics of a FIFO channel. It assumes that messages m and m' are sent on the same channel.

Our *delivered_all* construct is redundant for FIFO channels that are implemented using cumulative acknowledgments. When cumulative acknowledgments are used, an acknowledgment for a message with a given sequence number implicitly acknowledges the delivery of all messages with lower sequence numbers. The transport layer in the receiver will not acknowledge a message until all messages sent before it have been delivered. For a system that uses cumulative acknowledgments we can establish the following two facts:

$$delivered(id(m)) \Rightarrow delivered(id(pre_\sigma(m)))$$

$$delivered(id(m)) \wedge (last(\sigma) = m) \Rightarrow delivered\_all()$$

where $pre_\sigma(m)$ is the message preceding message m in the send sequence σ, and $last(\sigma)$

is the last message in σ. We can see that *delivered_all* is not as useful when cumulative acknowledgments are used since the same information can be obtained by performing a *delivered* on the last message sent. Although the information provided by *delivered_all* can be obtained through the use of *delivered*, it is still important to provide the *delivered_all* construct for systems that use cumulative acknowledgments. The use of *delivered_all* results in much more portable code which will work correctly irrespective of the acknowledgment scheme used by the underlying system.

2.3 Applying the Proof Rules

In this section we illustrate how our constructs and proof rules derived in previous sections can be used. We will examine a small two-process distributed program and show its correctness.

Consider a two-process system where one process represents a customer and the other process represents a bank. The customer process sends two successive messages to the bank. The first message represents a deposit and the second message represents a withdrawal. The customer must ensure that a positive balance is maintained in the bank; to ensure that the deposit is performed before the withdrawal, the customer executes a *delivered* call before sending the second message. The bank process simply receives the messages and updates the balance. The program is depicted in Figure 2.3. We will assume that the balance is initially set to 0. We would like to show that the program in Figure 2.3 does indeed ensure that the balance in the bank remains positive. Thus we need to prove that

$$I : bal \geq 0$$

```
C:: deposit = 20;                B:: receive(x);
    send(deposit) to B;              bal = bal + x;
    withdrawal = -15;                receive(x);
    delivered(mid(20));              bal = bal + x;
    send(withdrawal) to B;
```

Figure 2.3: Sample Program

is an invariant for the program. To prove invariant I, we give an annotated version of the program as depicted in Figure 2.4. Invariant I follows trivially if we can show our annotated program to be correct. As explained earlier, this involves three steps: a proof in isolation, a satisfaction proof, and a noninterference proof.

Showing that the annotations are consistent in isolation is trivial for both process C and process B. The *delivered* and *receive* statements are all miraculous in isolation. The correctness for the assignment and *send* statements are easily established. The process is illustrated by considering the first *send* statement in process C. From the send axiom, we must show

$$(\sigma = \emptyset \wedge deposit = 20) \;\Rightarrow\; (\sigma = \{20\})^{\sigma}_{\sigma \oplus deposit}$$

$$\equiv\; (\sigma = \emptyset \wedge deposit = 20) \;\Rightarrow\; \sigma \oplus deposit = \{20\}$$

which is trivially true. Correctness for the other *send* statement and the assignment statements can be shown in a similar fashion.

Next we need to establish satisfaction. Let us first consider satisfaction for the *delivered*

C:: $\{\sigma = \emptyset\}$
`deposit = 20;`
$\{\sigma = \emptyset \wedge$ deposit $= 20\}$
`send(deposit) to B;`
$\{\sigma = \{20\}\}$
`withdrawal = -15;`
$\{\sigma = \{20\} \wedge$ withdrawal $=$ -15$\}$
`delivered(mid(20));`
$\{\sigma = \{20\} \wedge hd(\rho_T) = 20$
$\wedge$ withdrawal $=$ -15$\}$
`send(withdrawal) to B;`
$\{\sigma = \{20, -15\} \wedge hd(\rho_T) = 20\}$

B:: $\{\rho = \emptyset \wedge$ bal $= 0\}$
`receive(x);`
$\{\rho =< 20 > \wedge$ x $= 20 \wedge$ bal $= 0\}$
`bal = bal + x;`
$\{\rho =< 20 > \wedge$ bal $= 20\}$
`receive(x);`
$\{\rho =< 20, -15 > \wedge$ x $=$ -15 $\wedge$ bal $= 20\}$
`bal = bal + x;`
{bal $= 5$}

Figure 2.4: Annotated Program

statement. We must show

$$(\sigma = \{20\} \wedge withdrawl = -15 \wedge 20 \in \rho_T) \quad \Rightarrow$$
$$(\sigma = \{20\} \wedge hd(\rho_T) = 20 \wedge withdrawl = -15)$$

The first and third conjunct of the consequent follows trivially. The second conjunct follows from the first and third conjunct of the antecedent and the network axiom, $(\sigma = \{20\} \wedge 20 \in \rho_T \wedge ms(\rho_T) \subseteq \sigma) \Rightarrow hd(\rho_T) = 20$. We must also show satisfaction for the two *receive* statements as

$$(\rho = \emptyset \wedge bal = 0 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \quad \Rightarrow$$
$$(\rho =< 20 > \wedge x = 20 \wedge bal = 0)^{x,\rho}_{MTEXT,\rho+MTEXT}$$
$$\equiv \quad (\rho = \emptyset \wedge bal = 0 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \quad \Rightarrow$$
$$(\rho + MTEXT =< 20 > \wedge MTEXT = 20 \wedge bal = 0)$$

and

$$(\rho =< 20 > \wedge bal = 20 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow$$
$$(\rho =< 20, -15 > \wedge x = -15 \wedge bal = 20)^{x,\rho}_{MTEXT,\rho+MTEXT}$$
$$\equiv (\rho =< 20 > \wedge bal = 20 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow$$
$$(\rho + MTEXT =< 20, -15 > \wedge MTEXT = -15 \wedge bal = 20)$$

The satisfaction formulae for the *receive* statements are not valid. The antecedents are not strong enough to show that the values received are 20 and -15 respectively.

As suggested by Schlichting and Schneider[90], we will strengthen our antecedents by the use of invariants. The following two invariants are easily established:

$$I_1 : \ (hd(\rho_T) = 20 \vee \sigma = \emptyset \vee \sigma = \{20\})$$
$$I_2 : \ (\sigma = \emptyset \vee \sigma = \{20\} \vee \sigma = \{20, -15\})$$

It is easy to see that I_1 and I_2 hold in process C. They are not affected by the actions in process B, and thus hold for process B as well; hence, I_1 and I_2 are invariants of the distributed program and can be used in the antecedents of the satisfaction formulae.

Adding I_1 to the antecedent of the satisfaction formula of the first *receive* we get:

$$(I_1 \wedge \rho = \emptyset \wedge bal = 0 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow$$
$$(\rho =< 20 > \wedge x = 20 \wedge bal = 0)^{x,\rho}_{MTEXT,\rho+MTEXT}$$
$$\equiv (I_1 \wedge \rho = \emptyset \wedge bal = 0 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow$$
$$(\rho + MTEXT =< 20 > \wedge MTEXT = 20 \wedge bal = 0)$$

The satisfaction formula for the first *receive* is now valid. The first disjunct of I_1 in combination with conjunct two and five of the antecedent implies that the value received is 20. The second disjunct of I_1 makes the antecedent false since it contradicts the fourth con-

junct of the antecedent. The third disjunct of I_1 in combination with the network axiom and conjunct two, four, and five of the antecedent again implies that the value received is 20.

Similarly, we can add I_2 to the satisfaction formula for the second *receive*:

$$\begin{aligned} & (I_2 \wedge \rho =< 20 > \wedge bal = 20 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow \\ & (\rho =< 20, -15 > \wedge x = -15 \wedge bal = 20)^{x,\rho}_{MTEXT,\rho+MTEXT} \\ \equiv\ & (I_2 \wedge \rho =< 20 > \wedge bal = 20 \wedge (\rho_T - \rho) \neq \emptyset \wedge MTEXT = hd(\rho_T - \rho) \Rightarrow \\ & (\rho + MTEXT =< 20, -15 > \wedge MTEXT = -15 \wedge bal = 20) \end{aligned}$$

The first two disjuncts of I_2 falsify the antecedent, since they contradict conjunct four. The third disjunct of I_2 in combination with the network axiom and conjunct two, four, and five implies that the value received is -15. The formula is valid and satisfaction is established.

The final step in our proof is noninterference, which follows trivially. To establish noninterference it suffices to observe that no assertions made in the sequential proof of one process involves a variable altered by the other process.

2.4 Chapter Summary

In this chapter we developed the formal framework needed to reason about transport layer information. A bipartite system model that lets us distinguish between delivery and receipt of messages was developed. As opposed to the traditional model used for reasoning about asynchronous communication, our model explicitly models the operation of the transport layer in the receiver. Based on the model we defined two constructs, *delivered* and *delivered_all*, which propagate transport layer information to the user level. The proof

rules for communication in the system as well as for our newly defined constructs were established. We illustrated with an example how the proof rules can be applied to program verification.

Chapter 3

Causally Ordered Communication

In the previous chapter we defined two sound and easily understood constructs, *delivered* and *delivered_all*, which reveal low-level information to the user. In this chapter we illustrate how our *delivered* and *delivered_all* constructs can be applied in practice. We present an implementation of causally consistent message passing based on our constructs. Causally ordered communication can simplify algorithm development, but has been hampered by its high implementation cost. Our implementation demonstrates how the additional system information provided by our constructs allows us to implement ordering constraints in a very different and more flexible way. We provide causally consistent message passing as a library routine at the user level. The efficiency of our proposed implementation as opposed to previously presented implementations will also be considered. Due to the flexibility of our constructs, they lend themselves to considering ordering constraints on a message-by-message basis. As an extension to causally consistent message passing, we will define causally ordered and causally early messages.

3.1 Introduction

In an asynchronous distributed system there is no notion of a global clock. No common physical time line exists, and thus, there exists no total temporal order on the events of the system. Instead, the events of a distributed system are only partially ordered based on their causality relation. The "happens-before" relation or causality relation, denoted $\rightarrow$, is an irreflexive partial order. As defined by Lamport[53], it is the smallest relation satisfying the following three conditions:

1. If e and e' are events in the same process and e occurs before e', then $e \rightarrow e'$.
2. If e is the sending event of a message by one process and e' is the corresponding receive event of that message in another process, then $e \rightarrow e'$.
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Clearly, $e \rightarrow e'$ indicates that event e may potentially affect event e'. Two events, e and e', which are not ordered by causality are said to be concurrent, $e \parallel e'$. Thus, $e \parallel e'$ if and only if $(e \not\rightarrow e') \wedge (e' \not\rightarrow e)$. The send event of a message m is denoted by $s(m)$ and the receive event of the message is denoted by $r(m)$.

Causally ordered message passing, or causally consistent message passing, ensures that all messages received at a process are received in the causal order they were sent. Concurrently sent messages may be received in any order. Causally ordered message passing was introduced by Birman and Joseph[13] in the context of broadcast communication. It was extended to point-to-point communication by Schiper et al.[89]. Causally ordered message passing can be formally defined as follows[23]. Let (s, r) denote corresponding send and receive events, meaning that s sent the message which was received by r. A computation is

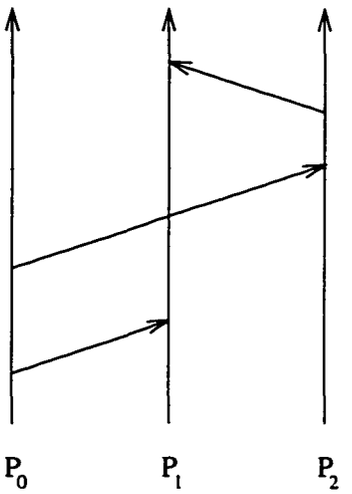

Figure 3.1: Causally Ordered Message Passing

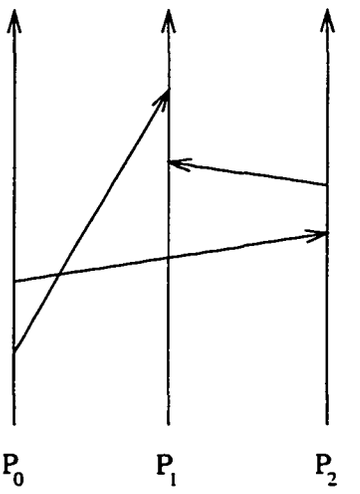

Figure 3.2: Non-Causally Ordered Message Passing

causally ordered if for all pairs of corresponding communication events in the computation, (s, r) and (s', r'):

$$r \sim r' \wedge s \rightarrow s' \Rightarrow r \rightarrow r'$$

where $r \sim r'$ indicates that r and r' are events in the same process.

The difference between a causally ordered message passing and a message passing that violates the causal order property is illustrated in figures 3.1 and 3.2. The message passing in figure 3.1 is causally ordered. The message passing in figure 3.2 is not causally ordered since the send of the second message received by P_1 causally precedes the send of the first message received. As illustrated by the figures, FIFO channels do not guarantee causal message passing. A good, less technical, discussion on causality and what causally ordered communication means can be found in [104]. It is easy to see why causally ordered message passing may be useful. Consider the computation in Figure 3.2. Let us assume that P_1

is a file server. Process P_0 sends a message to P_1 creating a file and then a message to P_2 notifying P_2 about the new file. In response P_2 sends a message to access the file at P_1. If the message passing is not causally ordered the message from P_2 may get to P_1 first, as depicted in Figure 3.2. As a result, P_2 will try to access the file before it exists at P_1. This is clearly an undesirable ordering of events, and a system in which such causally unordered communication is possible must explicitly implement a protocol to deal with this situation. In a system which provides causally ordered communication, such an ordering of events is not possible. The system will ensure that P_1 receives the message from P_0 before it receives the message from P_2. Another well-known, and less harmful, example of causal inconsistency comes from USENET news. It is often the case that replies to an article arrive at a site before the original article. The causal ordering between the articles are not preserved. Users of USENET news have simply learned to live with this artifact.

Causally consistent message passing in the broadcast or multicast context has been implemented in several systems: two of the most well-known systems being ISIS[14, 15] and Psync [73, 64]. The implementation in ISIS uses vector time to track the causal dependencies in the system. Based on the vector time of a message the receiver can determine if the delivery of the message to the application must be delayed. As described earlier, the implementation in Psync maintains a view of the context graph at each node. As in ISIS, the receipt of a message is delayed when there are causally preceding messages outstanding. Other proposed implementations are similar in nature[51, 96].

In a non-broadcast system the implementation of causally ordered communication is complicated by the fact that not every message is seen by every process in the system. The amount of information that needs to be included in each message to track causality in an N process system increases to $O(N^2)$ as opposed to $O(N)$ for a broadcast system. Two similar

implementations have been presented in the literature. The first implementation is based on vector time[89] whereas the second implementation uses message sequence numbers[83], a slight variant on vector time. As for the broadcast implementations, the extra information included in a message is used by the receiver to determine if the message needs to be delayed.

Due to the high cost involved in ensuring causal order a vast number of algorithms have been proposed for more efficient implementations based on specific system characteristics. Singhal and Kshemkalyani[92] take advantage of locality of communication. Assuming FIFO channels, only vector time entries which have been changed since the last message sent to a process need to be included in the next message to the process. Rodrigues and Verissimo[86] compress the information included in messages based on the communication topology, as represented by a graph. They identify a set of nodes, called causal separators, which can be used to filter causal information. Their work extends the idea of using gateways to filter causal information[62]. Causal order in a distributed system with a single server was studied by Kearns and Koodalattupuram[49]. Adelstein and Singhal [3] consider real-time multimedia systems. They focus on preventing causal ordering violations that are due to what they call "the triangle inequality"[1]. They argue that in a real-time system this is the only causal anomaly of importance; any message that is delayed for an extensive amount of time is discarded anyway. A similar idea is used by Ruget[87]. He cuts down on the cost of implementing vector clocks at the expense of providing less stringent ordering guarantees. In general however, the amount of piggybacked information needed to track causality grows linearly with the size of the system[22].

There is clearly a trade-off between the advantages of causally ordered communication and its implementation cost. Whether causal consistency should be provided by the com-

[1] Figure 3.2 illustrates the triangle inequality.

munication system or not has caused a heated debate in the distributed systems community over the past several years. The controversy was started by Cheriton and Skeene when they attacked the use of causally ordered communication[25]. Cheriton and Skeene argue that causal consistency should not be built into a system since it is expensive and there are many situations where it is not powerful enough. Based on the end-to-end argument[88] they argue in favor of a state-level approach to message ordering. The attack triggered replies from Birman [12], Reneese [105] and Cooper [26]. They claim that causally ordered communication can be implemented efficiently and that system development on top of it is greatly simplified. Development of the ISIS system has shown that causally ordered communication can be worthwhile.

All the implementations discussed above rely on the receiver to enforce the required causal ordering constraints based on extra information included in the messages. Our implementation, presented in the next section, takes a completely different approach. It relies on the sender to enforce the required causal order based on occasional synchronization. A similar idea was recently pursued by Mattern and Fünfrocken[61]. They suggest using two buffers, an input buffer and an output buffer, for each process. All message passing for a process goes through its associated buffers. The output buffers enforce the causal order by only allowing one outstanding message at a time. An output buffer does not transmit a new message until the previous message has been acknowledged by the receiving input buffer. The input buffers simply deliver messages to the application process in the same order they arrive. Our implementation does not use buffers, instead the sender uses our *delivered_all* construct to guarantee causal order. As we will see in the next section, our implementation of causally ordered communication provides causal consistency at the user level; hence, it allows causally ordered communication to be used on an application-by-application basis.

Rather than adding complexity and cost to the underlying communication system, our implementation provides causally ordered message passing through a library routine. The buffer processes suggested by Mattern and Fünfrocken could be implemented at the user level, but would then require user-level acknowledgments to be sent between the output and input buffers.

3.2 Implementation

Several proposed implementations of causally ordered communication were discussed in the previous section. The traditional approach to implementing causally ordered communication is to include additional information in each message and then require the receiver to enforce the appropriate ordering constraints. We take a different approach to the problem. Our implementation relies on the sender of a message to impose the required ordering constraints. No extra information is included in the messages. Instead, the sender uses the *delivered_all* construct[2], resulting in possible delay, to enforce the order. Our implementation provides causally ordered communication through the following library routine:

```
causal_send(data):      delivered_all();
                        send(data);
```

The correctness of our implementation is established by Theorem 1.

[2]The choice to use *delivered_all* is somewhat arbitrary. A *delivered* on the last message sent would work equally well. However, using *delivered_all* should be slightly more efficient.

Theorem 1 *The causal_send routine provides causally ordered communication.*

Proof: Consider two messages m and m' such that $s(m) \rightarrow s(m')$ and both m and m' are received by the same process. We must show that our implementation ensures $r(m) \rightarrow r(m')$. If $s(m) \rightarrow s(m')$, then there are two non-exclusive cases. The first possibility is that $s(m) \rightarrow s(m')$ because $r(m) \rightarrow s(m')$. For this case it follows trivially that $r(m) \rightarrow r(m')$. The second possibility is that both m and m' were sent by the same process or there exists a message m'' such that $s(m) \rightarrow s(m'') \rightarrow s(m')$ and m and m'' were sent by the same process. Without loss of generality let us assume that message m'' exists. Before sending message m'' our implementation calls *delivered_all.* On return from *delivered_all* we know that $\sigma = \rho_T$. Thus we know that message m was delivered before message m'' was sent. Since $s(m'') \rightarrow s(m')$, message m was also delivered before message m' was sent. But then by necessity, message m was delivered before message m'. Finally, from the Order Conservation Property we know that message m was received before message m' and hence $r(m) \rightarrow r(m')$. ∎

In this paper we are focusing on a system in which the communication subsystem provides reliable unordered message passing. If causally ordered communication is implemented on top of FIFO channels, then the implementation can be optimized to employ potential synchronization only when successive messages are sent on different channels:

```
causal_send(data) to P_i:     if(i != LAST){
                                 delivered_all();
                                 LAST = i;
                                 }
                              send(data);
```

The variable LAST records the channel on which the last message was sent and should be initialized to an illegal channel identifier. The implementation above assumes that a single FIFO channel connects any two processes. The transport layer in the receiver is still assumed to pass messages to the application in the same order they were delivered.

The major advantage of our implementation over previously presented implementations is that we can provide causally ordered communication at the user level. We do not need to build causally ordered message passing into the system, rather we can provide it as a library routine for the applications that need it. In light of the ongoing debate on whether causal consistency should be built into the system or not, a user-level implementation is a very appealing alternative. It is especially appealing considering that the debate on causally ordered communication was focused on a broadcast context where the overhead induced on each message is $O(N)$ for a N process system. When point-to-point communication is used the overhead on each message increases to $O(N^2)$. Requiring every message in the system to carry $O(N^2)$ extra information is a high price. Providing causally ordered communication at the user level for the applications that need it is a much more flexible approach. As described above, Mattern and Fünfrocken have suggested an implementation similar to ours. However, their implementation is not designed as a user-level implementation. Although their proposed buffer processes could be implemented at the user level, they are most naturally implemented at the transport layer. The potential advantage of having a buffer process, as opposed to the application process, enforce the required causal ordering constraints is that it may impose less delay on the application process.

3.3 Performance

The approach used in our protocol for casually consistent message passing stands in sharp contrast to the approach used in most previous implementations[89, 83]. Our approach uses occasional synchronization as opposed to including additional information in messages. In our protocol, the sender of the message must ensure that the ordering constraints are satisfied. The traditional approach is to require the receiver to enforce the ordering constraints, aided by the extra information included in the message. As mentioned earlier, a major advantage of our approach is that it is easily implemented at the user level. We do not need to build causal consistency into the system, rather we can supply it on an application-by-application basis. However, we must also consider the efficiency of our approach compared to previous implementations. This section will present a straightforward analysis to that measure. Executing a *delivered_all* call for every message may appear very costly at first. However, as we will see in this section, it is not as bad as it may appear for two reasons. First, even if we execute a *delivered_all* call every time we send a message, we do not necessarily have to synchronize every time we send a message. Second, no cheap alternative exists. In an N process system the traditional approach requires $O(N^2)$ extra information to be included in every message. The optimizations discussed earlier will not be considered in this section since they only apply for certain communication patterns or communication topologies. The implementation proposed by Mattern and Fünfrocken is also not considered.

For our analysis we will assume that network propagation time for one integer is unity. The length of a message is measured as a number of integers. As a convenience for our analysis we will consider the length of a message to be a continuous random variable. The

system will be characterized by the following parameters:

N Number of processes in the system.

α Network latency. This is the startup cost for a message. It is the time elapsed from when a process is ready to transmit a message until the message is actually starting to propagate over the communication channel. We will assume this delay to be constant, hence independent of the message size.

L Length of a message. L is a random variable with probability density function $f(L)$ and expected value $E(L)$.

S Intersend time. The intersend time is the time elapsed between two successive send operations. S is a random variable with probability density function $g(S)$ and expected value $E(S)$.

We will evaluate the expected delay for a message. The expected delay for a message is the expected time elapsed from when a process is ready to send a message until the message is available for receipt to the application in the receiving process. The implementation proposed by Schiper and Sandos and the one proposed by Raynal and Toueg are virtually equivalent with respect to efficiency. Both implementations require N^2 extra integers for a message. We will consider both implementations together as the "information approach". The expected delay for a message in our implementation will be denoted by T_S and the expected delay for a message in the information approach will be denoted by T_I. Establishing a lower bound for the expected message delay for the information approach is trivial:

$$T_I = \alpha + E(L) + N^2$$

The above equation is a lower bound on the delay since it ignores the potential *resequencing delay* encountered by the message at the receiver. If a message arrives out of causal order, then the receiver cannot make the message available for receipt until all causally preceding messages have arrived; hence, the message experiences a resequencing delay. In addition, the equation ignores the potential synchronization delay encountered by a message as discussed later.

The synchronization delay for a message is the delay encountered from when a sender is ready to request transmission of the message until the transport layer is ready to start servicing that request. A synchronization delay is experienced in our implementation due to the call to *delivered_all*. Let D be the synchronization delay for a message. D is a random variable with expected value $E(D)$. The expected message delay for our implementation can then be defined as:

$$T_S = \alpha + E(L) + E(D)$$

Comparing the equations for T_I and T_S we can see that when the quantity

$$\Delta = T_I - T_S = N^2 - E(D)$$

is positive our implementation has a smaller expected message delay than the information approach. To evaluate Δ we must further examine D.

The synchronization delay depends on both the message length and the intersend time. The random variable D can be formally defined as:

$$D = \begin{cases} 0, & \text{if } X - S < 0 \\ X - S, & \text{otherwise} \end{cases}$$

where $X = L+2\alpha+1$. The random variable X represents the total time elapsed from when we start transmission of a message until we know it has been delivered. The transmission time for a message of length L is $L+\alpha$. Assuming the receiver acknowledges each message individually the time to send an acknowledgment is $1+\alpha$. Hence, the total time from when we start transmission of the message until we know it has been delivered is $L+2\alpha+1$. If another message is sent during this time-interval, it will encounter a synchronization delay.

It is reasonable to assume that the length of a message is independent of the intersend time. Hence, we will assume L and S to be independent of one another. Based on our assumption of independence, we can represent the joint density of L and S by the product of their individual densities. The expected value of D, $E(D)$, can then be expressed as:

$$E(D) = \int_0^{\infty} \int_0^{l+2\alpha+1} f(l)g(s)(l+2\alpha+1-s)\,ds\,dl$$

To see how $E(D)$ compares to N^2 let us look at some specific examples. Let us assume that α has a value of 127. Measurements reported in Experiment 3, discussed in Chapter 7, indicate that this is a reasonable estimate for α; the constant communication overhead for datagram communication in our system is approximately 127 times larger than the cost associated with transmitting a single integer[3]. Let us first consider a system where both the message length and the intersend time are exponentially distributed. Initially, let L be exponentially distributed with mean 10. Figure 3.3 shows the square root of $E(D)$ as a function of the mean intersend time when the mean of the intersend distribution varies from 50 to 1000.

[3]The results for Experiment 3 suggest that the constant overhead for a message using our RUPP protocol is approximately 433 microseconds. The additional cost associated with each integer transmitted is approximately 3.42 microseconds.

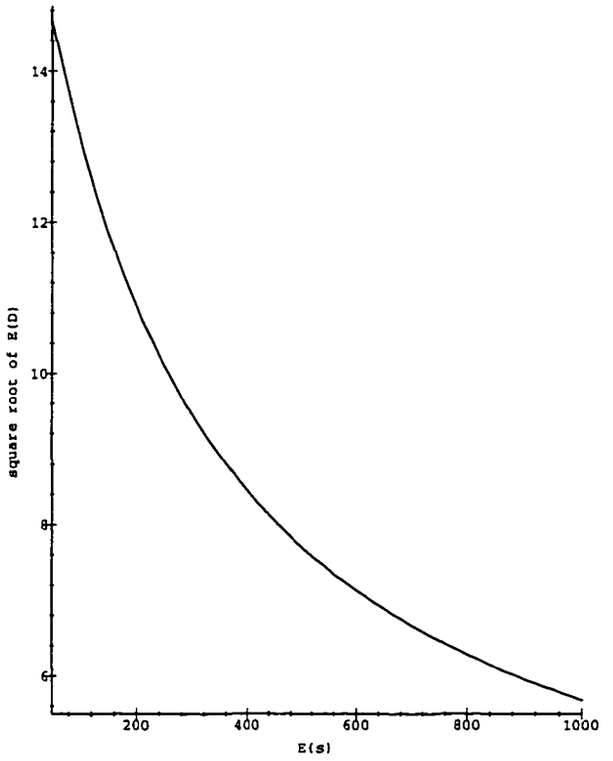

Figure 3.3: Sqrt($E(D)$) versus $E(S)$, L and S Exponentially Distributed.

A point $(E(S), \sqrt{E(D)})$ on the curve in Figure 3.3 gives the system size for which the expected delay for a message is the same in both our approach and the information approach, given the specific parameters used. For example, we can see that when the mean intersend time is 200 (roughly 700 microseconds) and the system size is 11 the delay for a message is approximately the same for our implementation and the information approach. The message delay in our implementation is independent of the size of the system. For the information approach on the other hand, the delay for a message is proportional to the square of the system size. As the system size grows the information approach becomes more

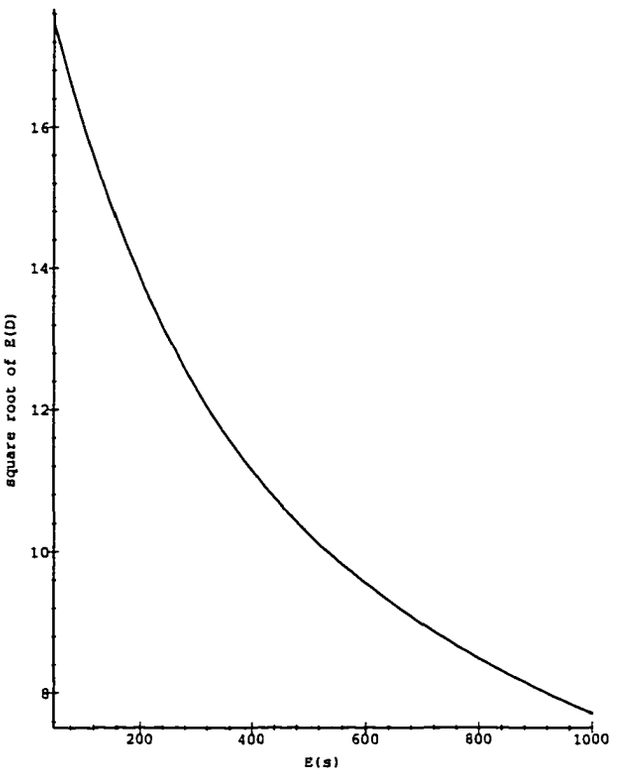

Figure 3.4: Sqrt($E(D)$) versus $E(S)$, L and S Exponentially Distributed, $E(L) = 100$.

expensive and our implementation compares more favorably.

To evaluate the influence of the mean message length, next let L be exponentially distributed with mean 100. The resulting graph is shown in Figure 3.4. As before, the intersend distribution is exponentially distributed with a mean varying from 50 to 1000. The graph in Figure 3.4 is very similar to the graph displayed in Figure 3.3. However, for larger size messages a slightly larger system size is needed for comparative delay between the two implementation methods. Larger messages take longer to transmit, and thus the synchronization delay for the *delivered_all* call increases. We can see in Figure 3.4 that when

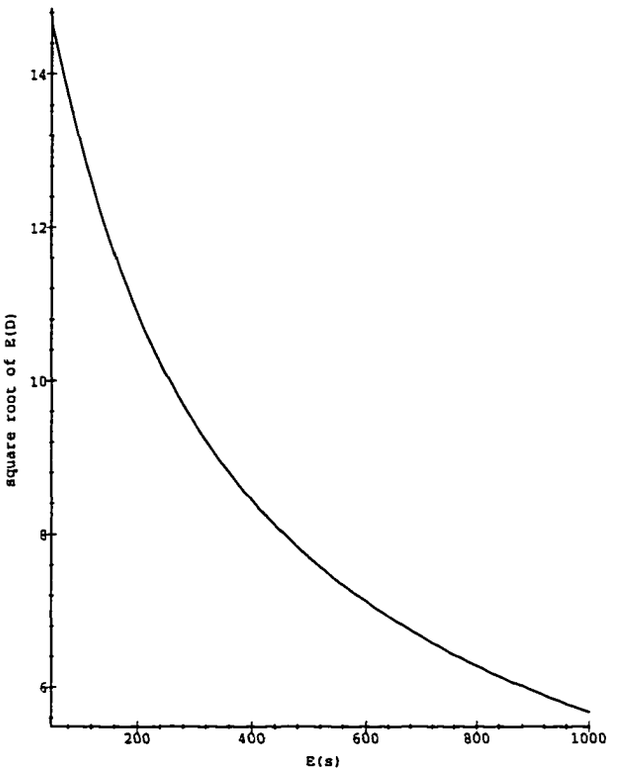

Figure 3.5: Sqrt($E(D)$) versus $E(S)$, L Constant, S Exponentially Distributed.

the mean intersend time is 200 a system size of approximately 14 processes results in equal delay for our implementation and the information approach.

The influence of the distributions of the message length and the intersend time should also be considered. Figure 3.5 displays the resulting graph when L has a constant size of 10, and the intersend time is exponentially distributed. Figure 3.6 displays the results for a system where L has a constant size of 10, and the intersend time is uniformly distributed between 0 and twice the mean, with the mean varying from 50 to 1000. As we can see from Figures 3.5 and 3.6, the results are not very sensitive to the particular distributions

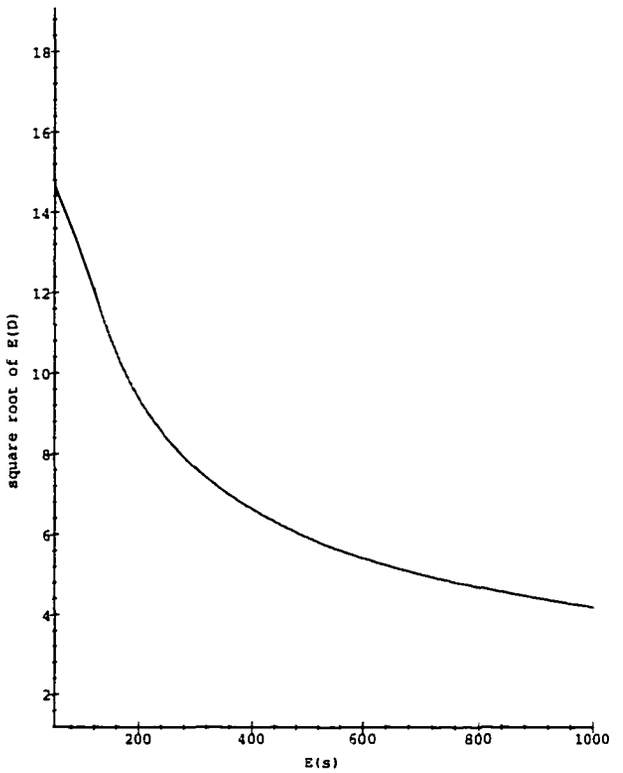

Figure 3.6: Sqrt($E(D)$) versus $E(S)$, L Constant, S Uniformly Distributed.

used. Although not displayed, we verified that the influence of mean message length on the distributions used for Figures 3.5 and 3.6 was consistent with the behavior displayed in Figures 3.3 and 3.4. For larger message sizes a slightly larger system size was needed to reach an equal delay between the synchronization and the information approach.

The results displayed in Figures 3.3, 3.4, 3.5, and 3.6 are very consistent. We can see from the figures that, due to the high cost of the information approach, our implementation performs well in relation to the information approach for medium and large sized systems.

There are some important factors which are not reflected in the graphs. Consider the

initial example above, a system of 11 processes with a mean intersend time of 200. For this system the mean message delay for a message sent using the information approach is 258 according to our equation for T_I. However, since the intersend time is smaller than the average time it takes to send a message, the networking support code may not be able to keep up with the application. The actual delay for a message sent using the information approach could be higher due to synchronization needed when the transport layer buffers are full. The information approach may suffer an additional synchronization delay for applications where the intersend time is smaller than the average message delay. Since our implementation has synchronization built into it, flow control and buffer overflow are not a consideration. However, we should be aware that synchronization performed when we send a message can impose an implicit delay on a later message since the execution of its send operation may be delayed. Since our implementation will impose a higher synchronization delay for most systems the hidden delay introduced by our implementation will be higher. This phenomenon will be investigated in more detail when we consider the performance of our flush channel implementation in the next chapter.

3.4 Message by Message Causal Ordering Constraints

Our implementation of causally ordered communication presented in the previous section allows causally ordered message passing to be used on an application-by-application basis. In this section we add additional flexibility and efficiency by applying the concept of causal consistency to individual messages. Causal order is traditionally seen as an ordering global to the system or at least a process group. However, for many applications, requiring every message to be causally ordered may impose unnecessary ordering constraints. Not all messages in an application are equally important or require the same ordering constraints. It

is therefore useful to apply causal ordering constraints to individual messages as opposed to an entire computation. We introduce the notion of a *causally ordered message.* Causally ordered messages are hard to implement efficiently. Fortunately, it is often sufficient to use *causally early messages* instead. Causally early messages are more efficient and are straightforward to implement using our *delivered_all* primitive.

Although causal order is generally viewed as a system-wide characteristic, it is not hard to think of an example where the causal order of a single message is really all we need. Consider a system of N processes, $P_0, \ldots, P_{N-1}$. Let process P_0 represent a bank, process P_1 represent a customer, and processes P_2 through P_{N-1} represent various stores. The bank process manages the customer process' accounts. Thus, P_0 receives and processes deposit messages from P_1 and withdrawal messages from the other processes. P_0 also sends the account balance to P_1 once a month. P_1 sends messages depositing money to P_0 and messages containing purchase orders to processes P_2 through P_{N-1}. Processes P_2 through P_{N-1} receive purchase orders from P_1 upon which they send a withdrawal message for the cost of the purchase to P_0. The customer makes a deposit at the beginning of the month and then performs multiple purchases throughout the month. One month the customer is running low on cash and wants to avoid a possible overdraft. The customer needs to ensure that the deposit message reaches the bank before any withdrawal messages associated with purchases performed after the deposit. The deposit message sent to the bank needs to be *causally ordered.* The other messages in the system have no ordering constraints. To require all messages in the system to be causally ordered, just because the customer occasionally needs to send a causally ordered deposit message to the bank, is very inefficient.

The above example illustrates the usefulness of causal order as applied to a single message. We must now formally define what it means for a message to be causally ordered.

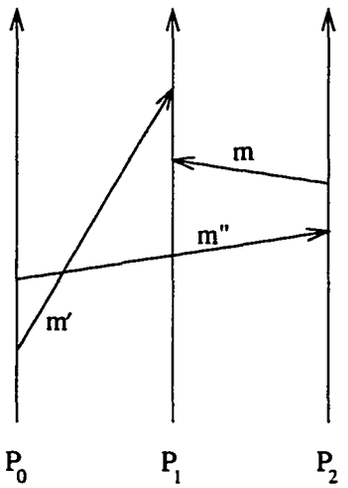

Figure 3.7: Problematic Computation

Let (s, r) denote the send and receive events of message m. Message m is causally ordered if for all pairs of corresponding communication events (s', r') in the computation:

$$(r \sim r' \wedge s \rightarrow s' \Rightarrow r \rightarrow r') \quad \wedge$$

$$(r \sim r' \wedge s' \rightarrow s \Rightarrow r' \rightarrow r)$$

Unfortunately, it is not possible to implement causal order for individual messages based on our primitives without requiring possible synchronization for every message; that is, implementing causally ordered message passing for the system. Consider the computation shown in Figure 3.7. Let us assume that process P_2 needs to send message m as a causally ordered message. Process P_2 has no way of locally ensuring that message m is not received before message m' based on our primitives. When causally ordered message passing is used, process P_0 would ensure that message m' is received before message m by executing

a *delivered_all* call before sending message m''. However, unless we have prior knowledge of the communication pattern in the system, process P_0 is not aware that process P_2 is about to send a causally ordered message. A process, P_i, cannot support causally ordered messages sent by other processes unless every message sent by process P_i is preceded by a call to *delivered_all*. Preceding every message by a call to *delivered_all* of course results in the system implementing causally ordered message passing.

The difficulty in implementing causally ordered messages is due to the second requirement of the definition for a causally ordered message; we must ensure that the message is not received before any message which causally precedes it. For most applications however, it is the first requirement in the definition which is most important; when an application sends a causally ordered message, it knows that it will be received before all messages which causally succeed it. Consider the banking example described above. A deposit request is sent as a causally ordered message to ensure that it reaches the bank before any withdrawal messages which causally succeed it. If the deposit message reached the bank before a message which causally preceded it, this would be perfectly fine. Hence, for the banking example it is sufficient to send a critical deposit request as what we will call a *causally early message*. Let (s, r) denote the send and receive events of message m. Message m is causally early if for all pairs of corresponding communication events (s', r') in the computation:

$$r \sim r' \wedge s \rightarrow s' \Rightarrow r \rightarrow r'$$

Causally early messages can be easily implemented locally based on our *delivered_all* construct. The sender of a causally early message must ensure that this message is delivered before it sends its next message. A regular send, *reg_send*, and a causally early send, *ce_send*,

can then be implemented by the following two library routines:

```
reg_send(data):     if(D_FLAG){
                       delivered_all();
                       D_FLAG = false;
                       }
                    send(data);

ce_send(data):      if(D_FLAG)
                       delivered_all();
                    D_FLAG = true;
                    send(data);
```

The sender maintains a flag to keep track of whether the last message sent was a causally early message or not. If the flag is set, then a call to *delivered_all* is performed before sending the next message. A call to *delivered_all* is performed prior to sending a message which succeeds a causally early message rather than at the end of the *ce_send* routine in order to minimize the delay imposed on the sender. The correctness argument for our implementation of causally early messages is similar to the correctness argument for causally ordered communication and is given by Theorem 2:

Theorem 2 *The reg_send and ce_send routines provide causally early messages.*

Proof: Consider two messages m and m' such that m is sent as a causally early message, $s(m) \rightarrow s(m')$, and both m and m' are received by the same process. We must show that our implementation ensures $r(m) \rightarrow r(m')$. If $s(m) \rightarrow s(m')$, then there are two non-exclusive cases. The first possibility is that $s(m) \rightarrow s(m')$ because $r(m) \rightarrow s(m')$. For

this case it follows trivially that $r(m) \rightarrow r(m')$. The second possibility is that m and m' were sent as successive messages by the same process or there exist a message m'' such that $s(m) \rightarrow s(m'') \rightarrow s(m')$ and m and m'' were sent as successive messages by the same process. Without loss of generality let us assume that message m'' exists. Since message m is a causally early message, D_FLAG will be set by the *ce_send* routine before message m is sent. Since m'' is the next message to be sent, D_FLAG will initially be set to true and our implementation will call *delivered_all*. On return from *delivered_all*, we know that $\sigma = \rho_T$. Thus we know that message m was delivered before message m'' was sent. Since $s(m'') \rightarrow s(m')$, message m was also delivered before message m' was sent. But then by necessity, message m was delivered before message m'. Finally, applying the Order Conservation Property we know that message m was received before message m' and hence $r(m) \rightarrow r(m')$. ∎

Sending individual messages as causally early messages when needed, as opposed to using causally ordered communication for the system, may greatly increase efficiency. Consider the banking example described above once more. Let us assume that k out of one hundred messages sent is a crucial deposit message. The expected synchronization delay for a message when causally ordered communication is used was derived in the previous section. Let us now consider what happens if we instead send the crucial deposit messages as causally early messages. The expected synchronization delay for messages immediately succeeding a causally early message will be as derived in the previous section. The expected synchronization delay for all other messages is zero. Fortunately most messages do not follow a causally early message. For our example, the expected synchronization delay for a message when causally early messages are used is $k\%$ of the expected synchronization delay for a message when causally ordered message passing is used in the system. Hence,

enforcing only the relevant causal ordering constraints can greatly increase efficiency for applications where k is small.

We believe the focus on causal order as a global concept is partially due to the approach used in previous implementations. When extra information is included in messages to track causality in the system, there is not much incentive to provide causal ordering constraints on a message-by-message basis. All the extra information must be included on every message even if we only occasionally need causal constraints. In our proposed implementation of causally early messages, on the other hand, the amount of synchronization needed can be proportional to the number of causal constraints imposed. Not only can our approach provide causal order on an application-by-application basis, it can also provide causal ordering constraints on a message-by-message basis.

An idea similar to our causally early messages has been pursued by Rodrigues and Verissimo[85] for a multicast context. They distinguish between two types of causal messages: opaque and transparent causal messages. The only restriction on a transparent message is that it cannot be received before an opaque message that causally precedes it. No messages are ever delayed waiting for a transparent message. Their suggested implementation still requires $O(N)$ extra information for all messages, including transparent causal messages. Their work is motivated by a need to cut down on the resequencing delay and to allow transparent messages to be sent without reliability constraints, issues which are more relevant for multicast communication than for point-to-point communication. For point-to-point communication, the resequencing delay is generally negligible compared to the delay induced by including extra information in the messages.

3.5 Chapter Summary

In this chapter we illustrated how our *delivered* and *delivered_all* constructs can be used in practice. We provided an implementation of causally consistent message passing. Causally consistent message passing can greatly simplify algorithm development, but also imposes a fairly high implementation cost. Whether causally ordered communication should be provided by the system or not is causing heated debate in the distributed systems community. Our implementation of causally consistent message passing provides the primitive in a user-level library. This alleviates the need for building expensive support into the underlying system and imposes the cost of causal order only on applications which use it. Our implementation is based on occasional synchronization as opposed to including extra information in messages. As an extension to causally consistent message passing this chapter also introduced causally ordered and causally early messages. Causally early messages can be implemented efficiently based on our *delivered_all* construct. Allowing causal ordering constraints to be applied on a message-by-message basis can radically improve performance. As illustrated by the banking example considered in this chapter, it is often sufficient to impose causal constraints only on selected messages.

Chapter 4

Flush Channels

The previous chapter described an implementation of causally ordered communication based on our *delivered_all* construct. As a second example of the applicability of our constructs this chapter will consider an implementation of *flush channels*. Flush channels offer a suitable alternative to FIFO communication for applications which require only a partially ordered message service. Our flush channel implementation utilizes both our *delivered* and our *delivered_all* constructs. As for causally ordered communication, the main advantage of our implementation is that we can provide it at the user level on an application-by-application basis. A simulation study of the performance of our implementation compared to previous implementations is also included in this chapter.

4.1 Introduction and Motivation

The Internet protocol stack, and traditional message passing primitives, provide either fully ordered connection-oriented service (TCP) or completely unordered (usually unreliable) service (UDP). However, for certain classes of applications, such as multimedia, a partially

ordered service is sufficient to ensure correctness and can greatly improve efficiency. Consider the transmission of a sequence of images across the network for display at a remote site. Each image may be split into several messages where each message contains part of the image data and a tag field identifying which part of the image it contains. The receiver assembles the image pieces into a frame and displays it. The situation described above creates a message pattern where we have batches of messages without any internal ordering constraints but where the batches need to be sequentially ordered. The pieces of one image may arrive in any order but we require all pieces of an image to arrive before parts of the next image. Requiring applications such as the one described above to use fully ordered service is overly restrictive. Flush channels [5] offer an appealing alternative for such applications. A flush channel allows the sender to enforce an arbitrary partial receipt order on messages. (A formal specification of flush channels is given in the next section.) Hence, a flush channel relaxes the total order, as appropriate for the application, to gain efficiency. Implementing the application described above on top of a flush channel would allow messages within a batch to arrive in any order while still enforcing the ordering constraint present between batches; this could amply improve performance compared to a FIFO channel. The performance benefits of flush channels over multi-link virtual circuits were considered in [18]. The performance benefits of a partially ordered as opposed to fully ordered transport layer protocol was also discussed in [8]. Here a partially ordered protocol where the partial order is negotiated between the sender and receiver prior to startup is considered.

As described above, flush channels offer great flexibility in the ordering constraints imposed on messages. However, the flush channel implementations suggested in the literature [6, 7, 48] are not as flexible. All implementations presented so far are most naturally imple-

mented at a low level by building flush channels into the underlying communication system. Although flush channels are very useful for some applications, it is not realistic to expect the majority of applications to use flush channels. Flush channels are not a standard construct which we can expect the communication subsystem to provide. Building flush channels into the underlying communication system is also inflexible and results in added networking complexity. In this chapter we provide a user-level implementation of flush channels. Providing flush channels at the user level is the missing link in allowing flush channels to be a viable alternative for applications which can benefit in performance by relaxing the total ordering on messages. Based on our *delivered* and *delivered_all* primitives we provide flush channels as a set of library routines. These library routines can be incorporated by applications which use flush channels but impose no cost to other applications. As our performance study will show, our implementation performs most favorably when the ordering requirements are weak and messages are short.

4.2 Flush Channel Semantics

Flush channels were introduced by Ahuja[5]. As mentioned earlier, a flush channel allows the sender to construct an arbitrary partial order, specifying constraints on the receipt order of the sent messages. A message sent on a flush channel has a *type* in addition to the data. The partial order is built by sending messages of different types. There are four different types of messages, each one imposing a different ordering constraint:

- Type **Ord**, an *ordinary* message. An ordinary message imposes no ordering constraint. However, an ordering constraint can be be imposed on the message by a message of another type.

- Type **2F**, a *two-way flush* message. A two-way flush message must be received after all messages sent prior to it, and before all messages sent after it.
- Type **FF**, a *forward flush* message. A forward flush message must be received after all messages sent prior to it.
- Type **BF**, a *backward flush* message. A backward flush message must be received before all messages sent after it.

The proof rules for flush channels were developed by Camp, Kearns and Ahuja[19]. Their development is based on Schlichting and Schneider's work on proof rules for asynchronous communication[90] discussed in Chapter 2. The state of a flush channel, F, is modeled through implicit variables. When a message is sent, it is inserted into the send multiset σ_F, and when a message is received, it is inserted into the receive multiset ρ_F. The ordering restriction imposed on F is modeled by an irreflexive partial order, $\prec\!\!+_F \subseteq \sigma_F \times \sigma_F$. For $m, m' \in \sigma_F$, $m \prec\!\!+_F m'$ if and only if m must be received before m'. The partial order is implicitly constructed by the sender as new messages are sent along the channel. The semantics ensured by a flush channel can then be expressed by the following Flush Channel Network Axiom:

Flush Channel Network Axiom: For flush channel F, the following two properties must hold:

En Route Property: $\rho_F \subseteq \sigma_F$

Order Property: $\forall m, m' : m, m' \in \sigma_F \wedge m \prec\!\!+_F m' \wedge m' \in \rho_F \Rightarrow m \in \rho_F$

The En Route Property simply says that we cannot receive a message before it is sent. The Order Property says that messages must be received in an order that is consistent with

the partial order specified by the sender. The semantics of a flush channel given by the Flush Channel Network Axiom must be ensured by any implementation of flush channels. The En Route Property, of course, follows directly from the semantics of any underlying communication channel and is of no concern.

4.3 Flush Channel Implementation

Having formally defined flush channels in the previous section we now turn to the matter of implementation. Our implementation provides flush channels through a *flush_send* library routine. The *flush_send* library routine expands a send for each one of the four message types into sequences of *delivered*, *delivered_all*, and *send*s on the unordered communication system. Based on the type of the message a *flush_send* call is expanded as follows:

```
flush_send(2F, data):   delivered_all();
                        send(data);
                        D_FLAG = true;
                        mid = id(data);

flush_send(FF, data):   delivered_all();
                        send(data);
                        D_FLAG = false;

flush_send(BF, data):   if(D_FLAG)
                           delivered(mid);
                        send(data);
                        D_FLAG = true;
                        mid = id(data);
```

```
flush_send(Ord, data): if(D_FLAG) {
                           delivered(mid);
                           D_FLAG = false;
                           }
                       send(data);
```

As can be seen from above, our implementation uses occasional user-level synchronization, through the use of *delivered* and *delivered_all*, to ensure the necessary message ordering. The boolean D_FLAG is used to indicate whether the next message needs to be potentially delayed or not. The message identifier of the message must also be saved so that the next flush_send can identify the message for whose delivery it must wait. Recall that the $id(\cdot)$ function is defined to return the message identifier for the message. Note that D_FLAG is false at the completion of a send event for an ordinary message. Thus only the first message in a batch of ordinary messages can ever require synchronization. To show that our implementation is correct we must show that the Order Property of the Flush Channel Network Axiom is maintained.

Theorem 3 *The flush_send routines ensure the Order Property of the Flush Channel Network Axiom.*

To prove Theorem 3 we will first establish that our implementation ensures the weak order property:

Lemma 1 *The flush_send routines ensure the Weak Order Property:*

$\forall m, m' : m, m' \in \sigma \wedge m \prec +_F m' \wedge m' \in \rho_T \Rightarrow m \in \rho_T$

Proof: To ensure the Weak Order Property we must guarantee that all messages sent before a two-way flush or forward flush message are delivered before this message. This is ensured in our implementation by executing a *delivered_all* call prior to sending a two-way

flush or forward flush. On return from *delivered_all* we know that $\sigma = \rho_T$. All messages sent so far have already been delivered and thus are delivered before the two-way flush or forward flush we are about to send.

We must also guarantee that all messages sent after a two-way flush message or backward flush message are delivered after this message. This is ensured in the implementation by delaying a message which immediately follows a two-way flush or a backward flush until the two-way flush or backward flush has been delivered. If the message immediately following is a two-way flush or a forward flush this is ensured by the stronger guarantee that all messages sent prior to it have been delivered as discussed above. For an ordinary message or a backward flush this is ensured by executing a *delivered* call on the message identifier of the two-way flush or backward flush. On return from *delivered* we know that the indicated message is in ρ_T. The two-way flush or backward flush message has been delivered and thus the message we are about to send and all successive messages will be delivered after the two-way flush or backward flush. This establishes Lemma 1. ∎

The proof of Theorem 3 follows directly from Lemma 1 and the Order Conservation Property. Lemma 1 ensures that all messages are delivered in a correct order at the transport layer of the receiver, and the Ordering Conservation Property ensures that the messages are received in the same order at the application layer.

4.4 Previous Implementations

Our implementation of flush channels should be compared to other implementations previously presented in the literature [6, 48, 7]. The earliest implementation described in the literature is based on "selective" flooding[6]. A flush message is implemented by sending a message on every possible path between the sender and the receiver. The technique re-

quires the underlying network to be reliable and assumes each intermediate switch node to have FIFO queues for all incoming and outgoing channels. When a flush message arrives at a switch node, the node is responsible for placing the flush message in the queues of all outgoing channels which lead to the destination. This way a copy of the flush message can be transmitted over every channel between the sender and the receiver. The decision on when a flush message can be passed on to the application depends on the type of message transmitted. A two-way flush or forward flush message can be received by the application when the message is at the head of the queues of all incoming channels. This ensures that all messages sent before the two-way flush or forward flush has been received. A two-way flush at the head of the queue blocks all messages behind it, whereas messages behind a forward flush are still eligible for receipt. A backward flush can be received as soon as the first copy arrives on an incoming channel. A backward flush blocks messages behind it in the queue. This way the backward flush is received before all messages sent after it. Ordinary messages are transmitted over a single path in the network. The flooding technique is not very practical. For a large network with many paths between a sender and a receiver a large number of messages is necessary. For two-way flush and forward flush messages it is necessary to wait for the slowest copy of the message before passing the message to the application. In addition, the implementation is inflexible in that it requires all nodes along any path between two communicating nodes to support flush channel communication.

An improved implementation, termed the "waitfor" technique was suggested by Camp, Kearns and Ahuja[48]. In the waitfor technique each message contains two integers: a sequence number and a waitfor value. The transport layer at the receiver can determine the order in which messages must be presented to the application based on the waitfor value and the type of the message. For a two-way flush or forward flush message the

sender sets the waitfor value to the sequence number of the message preceding the flush message. The receiver knows that the message cannot be handed to the application until all messages with sequence numbers smaller than or equal to the waitfor value have been received. For a backward flush or ordinary message the sender sets the waitfor value to the sequence number of the current *backward flush point*, where the backward flush point is the backward flush or two-way flush message last sent. The receiver knows that the message cannot be received until after the message with a sequence number equal to the waitfor value has been received. When buffer space is limited, as in any real implementation, dummy messages may be required to synchronize the sender and the receiver and avoid deadlock[18]. One additional implementation, very similar in nature to the waitfor technique, has been suggested in the literature[7]. This technique also works by including two extra integers in each message from which the receipt order can be derived. The two techniques have been shown functionally equivalent[18].

The implementations described above take a very different approach from our implementation. In our implementation, synchronization may be needed when a flush message is sent. Ordinary messages are sent without extra overhead unless they immediately follow a two-way flush or backward flush. In contrast, the waitfor technique and its cohort imposes overhead (in the form of piggybacked ordering information) on every message *even when no resequencing delay is encountered.* For most applications using flush channels, we expect the majority of the messages transmitted to be ordinary messages. The flush messages would be used to separate, terminate, or indicate a batch of ordinary messages[19]. In this scenario we believe our implementation to be competitive in terms of performance. The locations in the application where flush messages are sent can be seen as synchronization points which are naturally represented in our implementation. As mentioned previously,

the major advantage of our implementation is that it supplies flush channels through library calls rather than as part of the underlying communication system. Only applications that use flush channels will encounter an overhead. In addition, the amount of overhead experienced is directly proportional to the amount of order imposed on the receipts of messages. The higher the ratio of flush messages to ordinary messages, the greater the overhead. A performance study of our implementation compared to the waitfor technique will be given in the next section.

4.5 Performance

A straightforward implementation of flush channels based on our *delivered* and *delivered_all* constructs was provided in an earlier section. In this section we consider the performance of our proposed implementation relative to implementations previously given in the literature. We will focus on the comparison with the waitfor technique [48], which we believe is the most competitive implementation presented so far. Our performance evaluation will be based on a simulation study. In our comparison, we should keep in mind that our proposed user-level implementations do not necessarily have to outperform previous implementations. The major advantage of our flush channel implementation is its greater flexibility.

The performance of "batched flush channel applications" under various system parameters and in comparison to multi-link virtual circuits has been investigated for the waitfor technique by Camp[18]. The average delay for a message was evaluated through both simulation and analysis, where the analytic results are an extension of the analysis on resequencing delay in an M/M/m system[4]. The performance system model for the waitfor technique is illustrated in figure 4.1. A sender S and a receiver R are communicating over m physical links. An incoming message is transmitted if there is a link available, otherwise

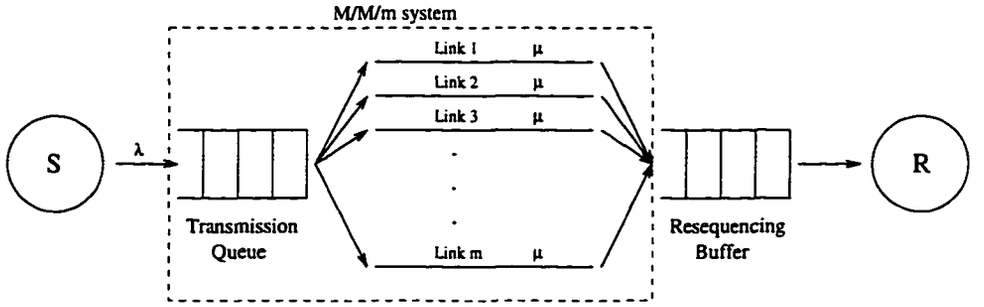

Figure 4.1: Performance Model for the Waitfor Technique

it is buffered in the transmission queue. Once a message reaches its destination, it might experience a resequencing delay. The message must be buffered until all messages which must be received before it have arrived. A message must also be buffered until the application executes a *receive* statement but this delay is not considered when evaluating the performance. We are only interested in the delay from when the message was sent until it is eligible for receipt by the application. The expected delay for a message is found by adding up the expected delay in the transmission queue, the expected transmission time and the expected resequencing delay. As indicated in figure 4.1, the system excluding the resequencing buffer is an M/M/m queue. The average delay of an M/M/m queue is well known [50] and is independent of the imposed ordering constraints. The resequencing delay, on the other hand, depends on the ordering constraints and was derived individually for each considered application[18].

As illustrated in figure 4.2, our performance system model will look slightly different from the one presented by Camp. In our implementation a message need never be delayed at the receiver, and thus there is no resequencing buffer in our model. Unfortunately, our system is not an M/M/m queue due to constraints on when a message can be transmitted.

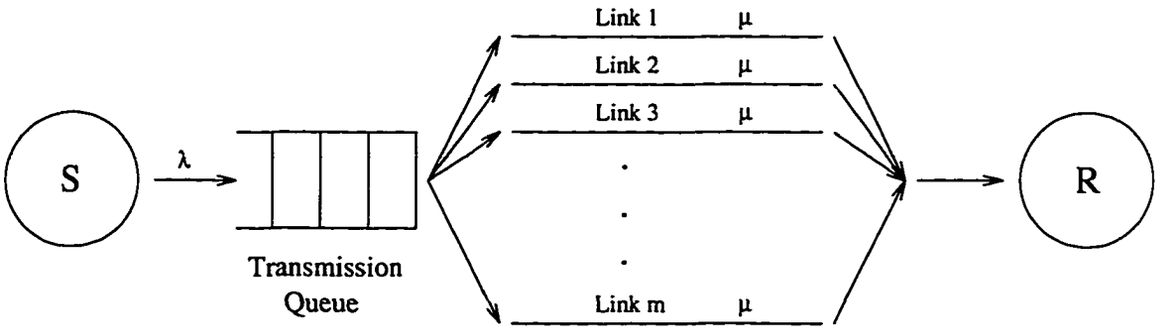

Figure 4.2: Performance Model for Our Implementation

Analysis of the system is complicated by the fact that a message might have to be delayed even when there are empty links in the system. In this dissertation we focus on a simulation study comparing the two implementations.

Simulating flush channel communication for our implementation as well as for the waitfor technique is a straightforward extension to an M/M/m queue simulation[69]. We will focus on a batched flush channel application where the batches are separated by two-way flush messages. Our simulations used the method of batch means to calculate 95% confidence intervals of the expected delay for a message. The widths of the confidence intervals were all within 2% of the values calculated for their corresponding point estimates. For notational convenience, only the point estimates are displayed in our graphs. As explained above, the delay for a message is the time elapsed from when the message is transmitted until it is available for receipt by the application. For all our simulations the number of links between the sender and the receiver was set to 25. The intersend time between messages as well as the transmission time was assumed to be exponentially distributed with means $1/\lambda$ and $1/\mu$ respectively. Since the waitfor technique includes an extra integer in each message the average transmission time was set to be slightly higher for the waitfor technique than for

our approach. We use μ_i to indicate the transmission rate for the waitfor implementation and μ_s to indicate the transmission rate for our implementation.

Figure 4.3 shows the average message delay as a function of λ when there are 999 ordinary messages in a batch. The curve for our implementation is labeled "synch" for synchronization approach and the curve for the waitfor technique is labeled "info" for information approach. The transmission rates were set at $\mu_i = 0.1$ and $\mu_s = 0.1111$. Assuming that the time to transmit one integer is unity a transmission rate of 0.1 corresponds to a average message size of 10 integers and a transmission rate of 0.1111 corresponds to a average message size of approximately 9 integers.

We can see from Figure 4.3 that as expected the expected message delay increases with λ. However, we can see that, for our approach, the delay does not increase at all as fast as for the information approach. We can see that, when λ goes above 2.25, the expected delay for a message sent using the waitfor technique increases dramatically. This is due to a very high queuing delay in the transmission queue. A send rate of 2.25 corresponds to a link utilization of 0.9. Our implementation does not experience the same build-up in the transmission queue. In our implementation, the two-way flush messages work as a natural form of flow control. Before sending a two-way flush message, *delivered_all* is called. The application pauses until all messages sent prior to the two-way flush have been delivered and the transmission queue is empty. After the two-way flush has been delivered the messages in the next batch can be transmitted without hardly any delay besides their actual transmission time.

As illustrated by Figure 4.3, the expected delay for a message is lower for our implementation than for the waitfor technique. However, our occasional synchronization introduces a hidden delay for a message. When the application pauses, no messages are generated, which

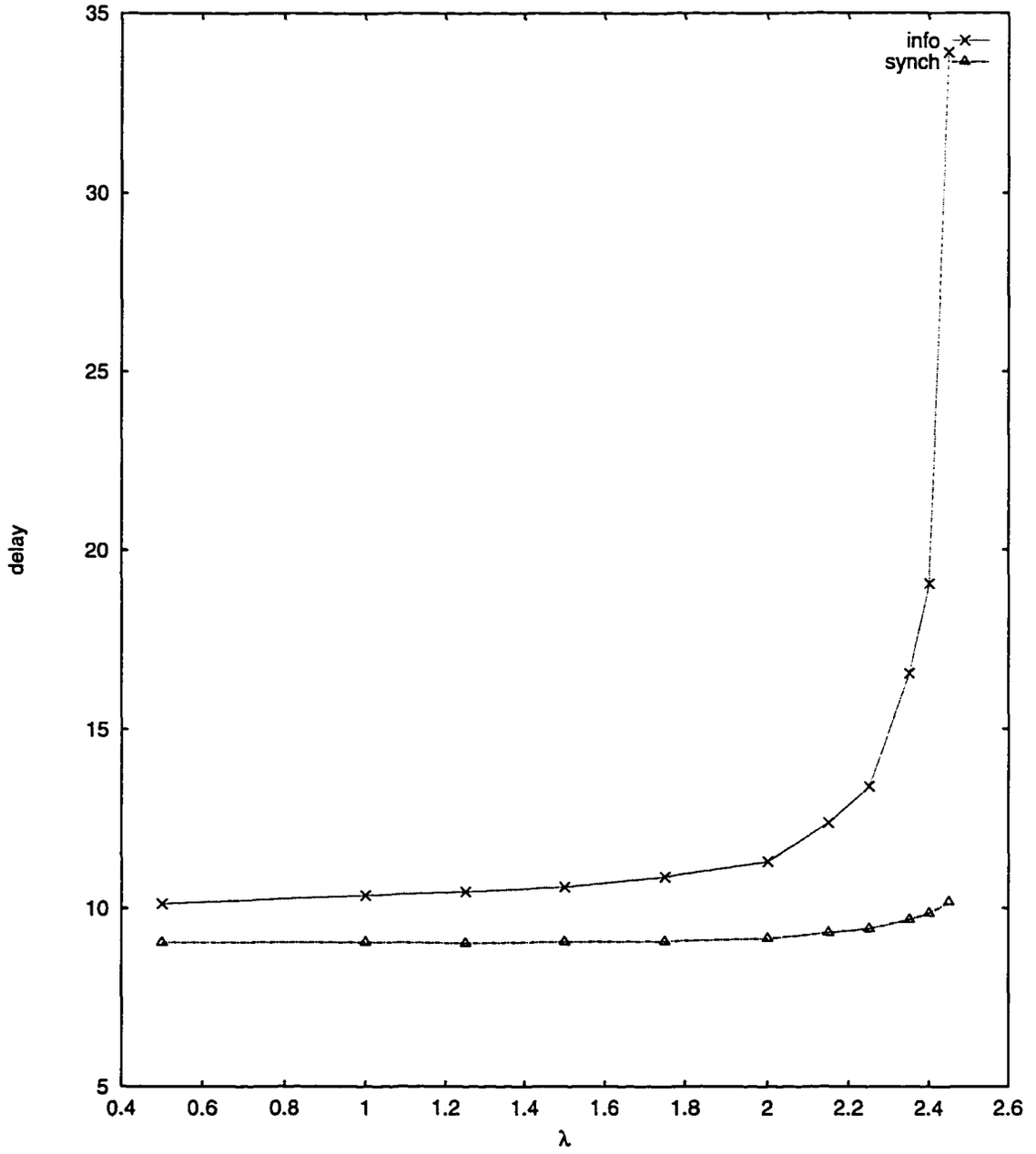

Figure 4.3: Expected Message Delay versus λ

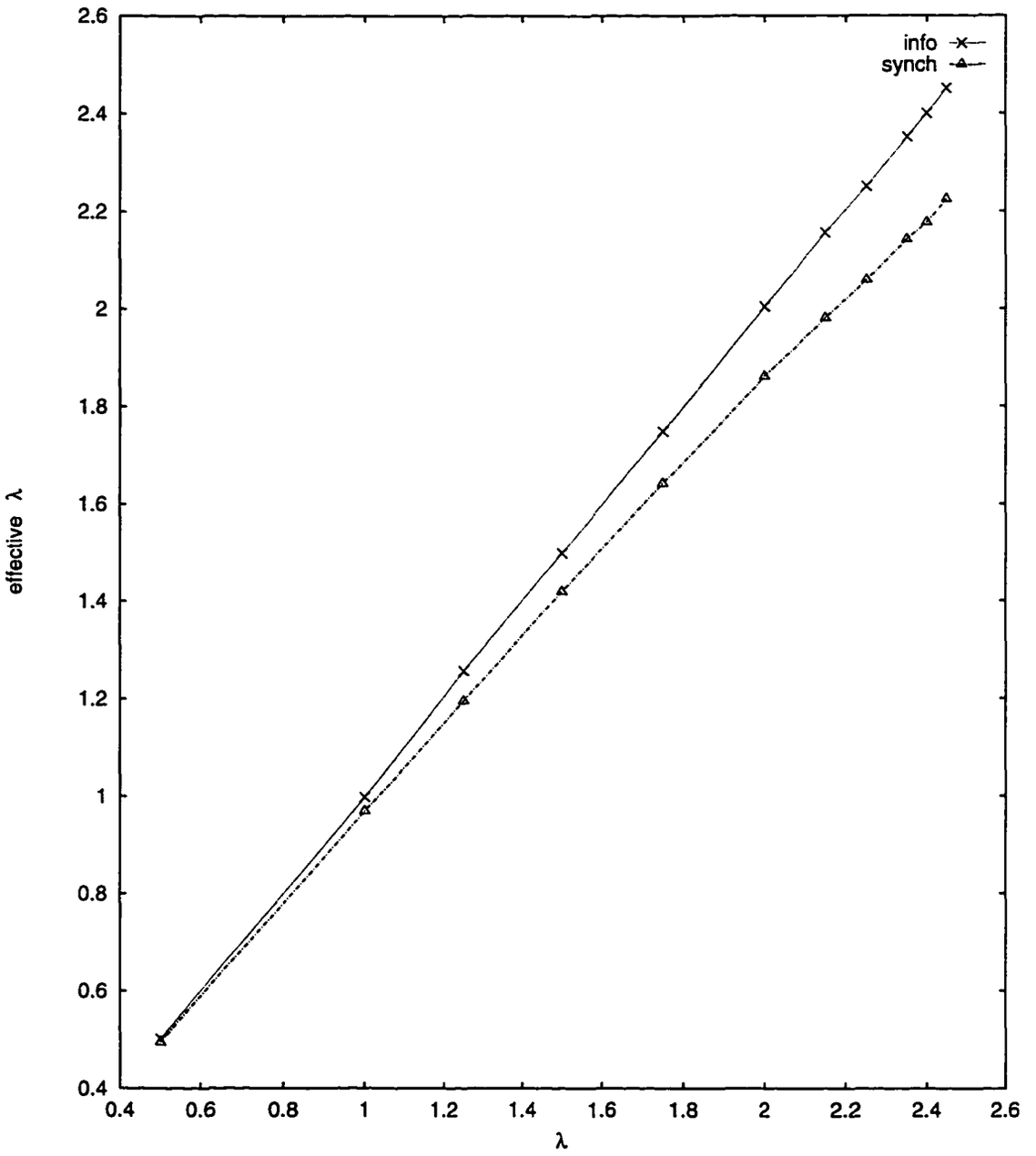

Figure 4.4: Effective Send Rate versus λ

causes the actual send of a later message to be delayed. Hence, to be fair in our analysis we must also examine the "effective send rate" for the two applications. The effective send rate versus λ is shown in Figure 4.4 for the experiment from Figure 4.3.

We can see from Figure 4.4 that the effective send rate for our implementation is slightly lower than λ. As λ increases, more delays are imposed by our implementation, and the effective send rate starts to deviate more from λ. As expected, the effective send rate matches λ for the waitfor technique. Since our simulation assumes an infinite transmission buffer the waitfor technique imposes no send delay. In practice the transmission buffer would be finite, and a send operation would block when the transmission buffer was full. This would cause the effective send rate for the waitfor technique to deviate from λ for high utilizations.

Realizing that the effective send rate differs between the two implementations, it is important to consider delay as a function of the effective send rate. This information is displayed in Figure 4.5. We can see from the figure that, the delay imposed by our implementation is lower than the delay imposed by the waitfor technique even when the effective send rate is considered. The graph for the waitfor technique is of course identical to the graph in Figure 4.3. (Again, this is due to our use of an infinite send buffer.) The graph for our implementation now show that the delay for our implementation increases rapidly when the effective send rate gets very high. Even though the two-way flush messages work as flow control, the queuing delay in the transmission queue increases when the effective send rate becomes high.

Next, let us examine the influence on performance by the batch-size. Figure 4.6 displays the expected delay for a message as a function of the batch-size. The send rate was held constant at 2.0 and all other parameters were the same as in the previously described

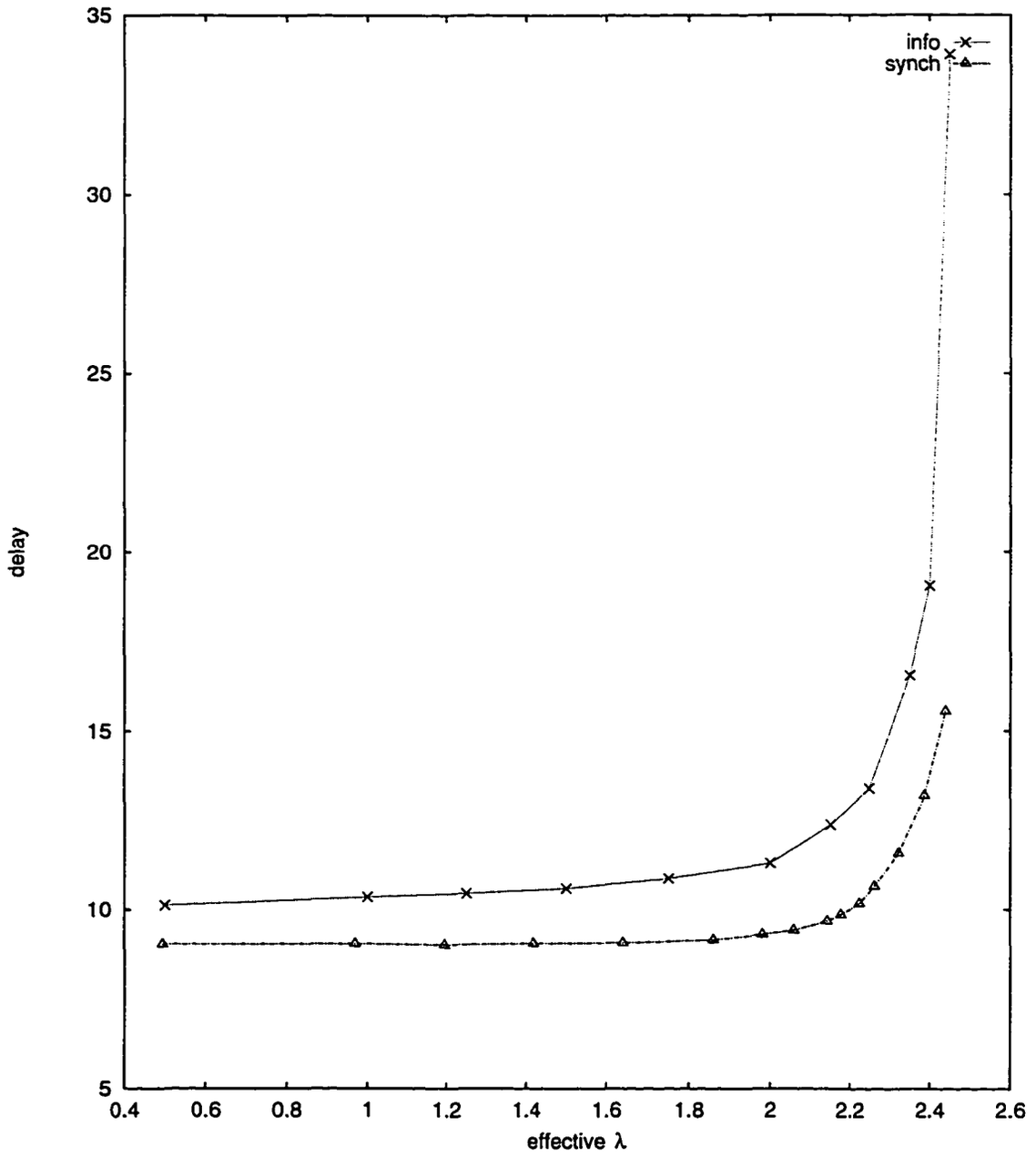

Figure 4.5: Expected Message Delay versus Effective λ

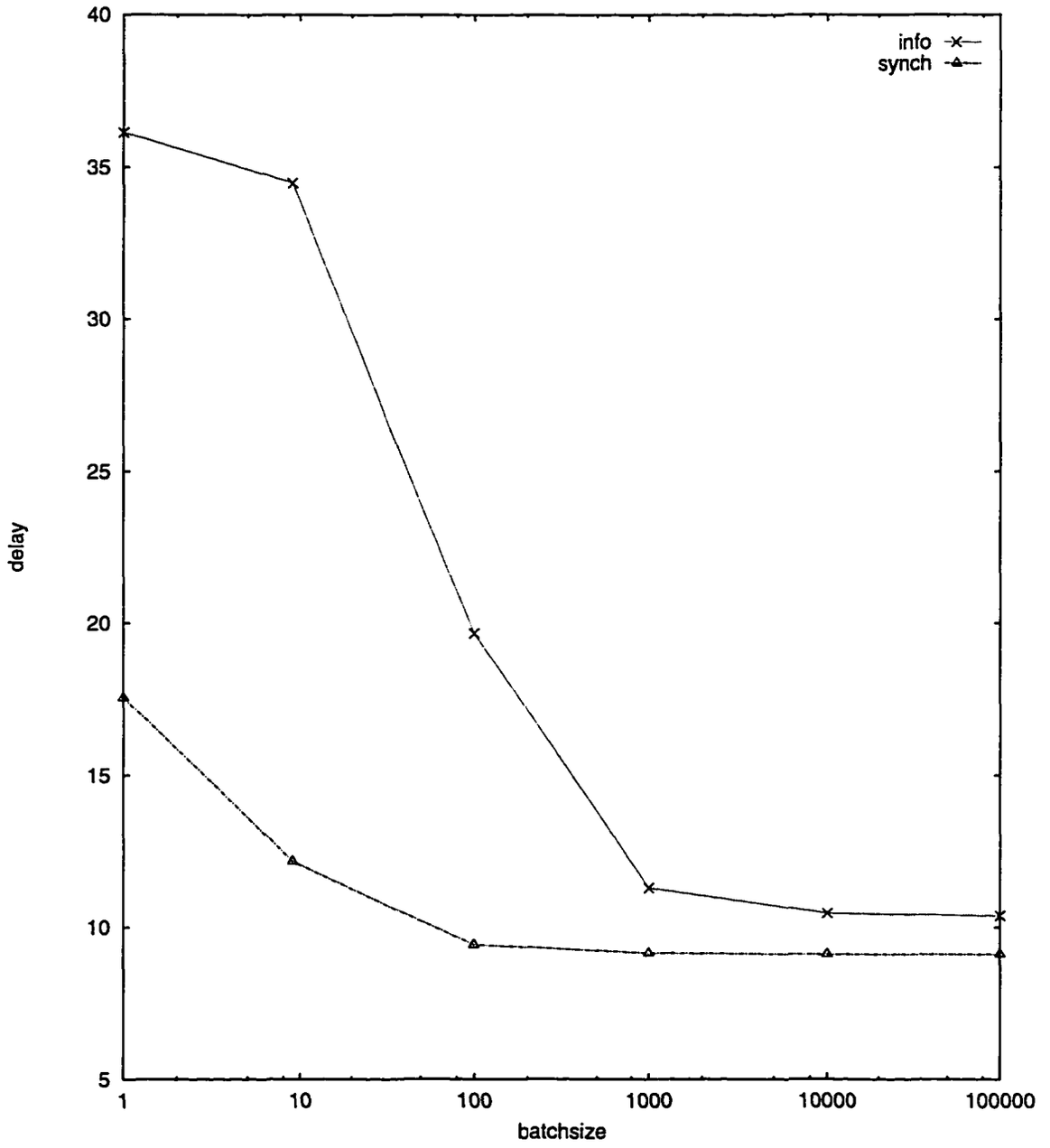

Figure 4.6: Expected Message Delay versus Batch-size

simulation. As expected, we can see that the average delay for a message decreases as the batch-size increases for both implementations. As the batch-size increases less two-way flush messages are transmitted. Fewer flush messages leads to less synchronization for our implementation and a smaller resequencing delay for the waitfor implementation. Again, we can see that the expected delay for a message is smaller in our implementation than in the waitfor implementation. However, we must also consider the "hidden delay" discussed above by examining the effective send rate. Figure 4.7 shows the effective send rate as a function of batch-size. We can see from Figure 4.7 that our implementation is not a good solution for a batch-size of one, even though the expected delay for a message is significantly smaller than for the waitfor implementation. When a batch-size of one is used every message requires potential synchronization. As a result the effective send rate for our implementation is very low. As before, an infinite transmission buffer is assumed and the send rate for the waitfor technique stays constant at λ. In practice the transmission buffer is finite, and a send may block to avoid buffer overflow.

Finally, let us examine the influence of message length on the relative performance of the two implementations. The average length of a message is reflected in the difference in μ_i and μ_s. The value used for μ_s above, 0.1111, was based on a μ_i value of 0.1 and a message length of 10. If the message length was instead 5, then the corresponding value for μ_s would be 0.125. Hence, to examine the influence of message length we will fix μ_i and then calculate μ_s based on varying message lengths. Figure 4.8 shows the average message delay versus the message length used to calculate μ_s. The batch-size was set to 999 and λ was set at 2.0. As illustrated by Figure 4.8 our implementation is more beneficial for short messages relative to the waitfor implementation. For short message the extra information included in a message by the waitfor technique is more costly. If the messages are long,

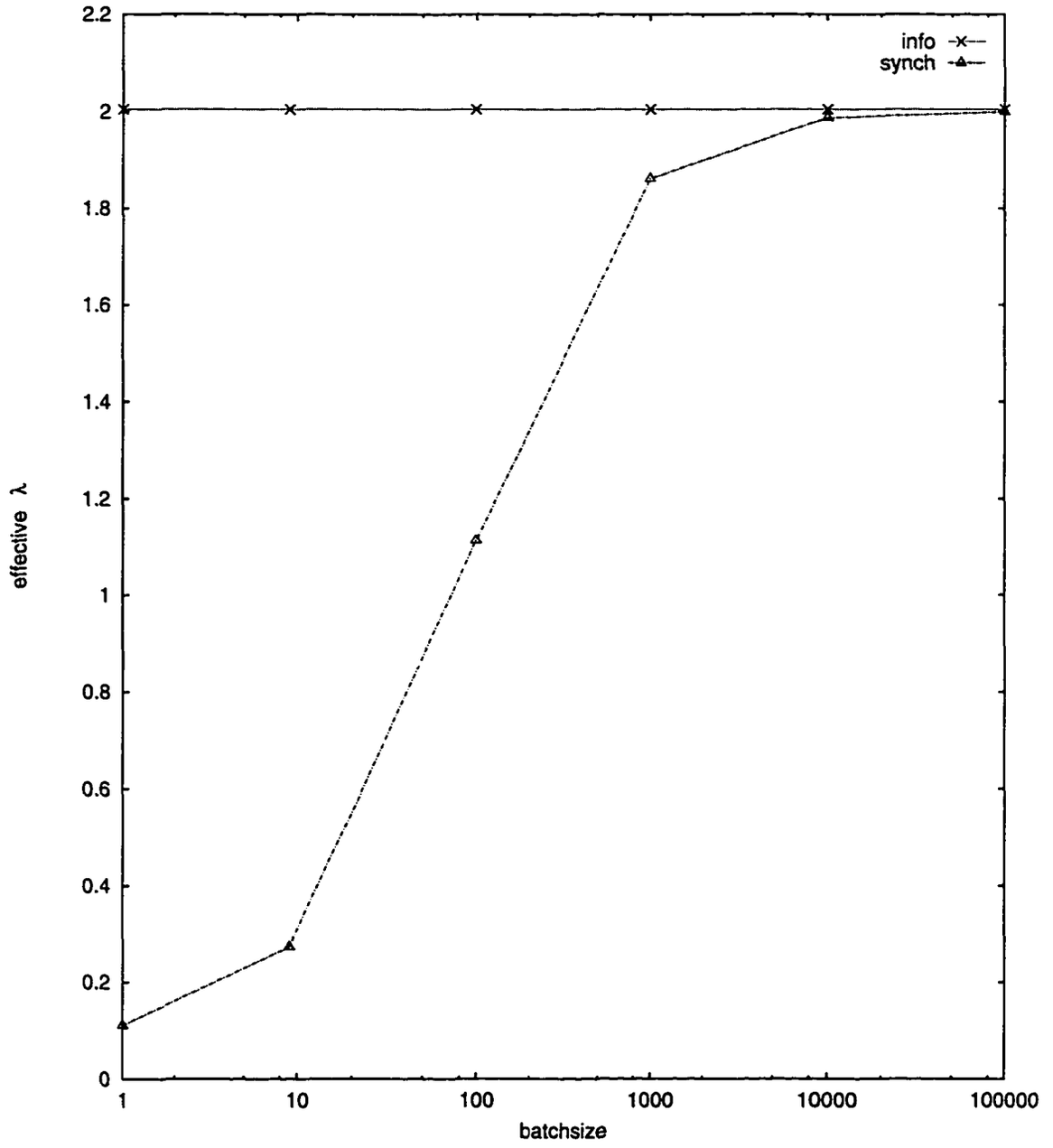

Figure 4.7: Effective λ versus Batch-size

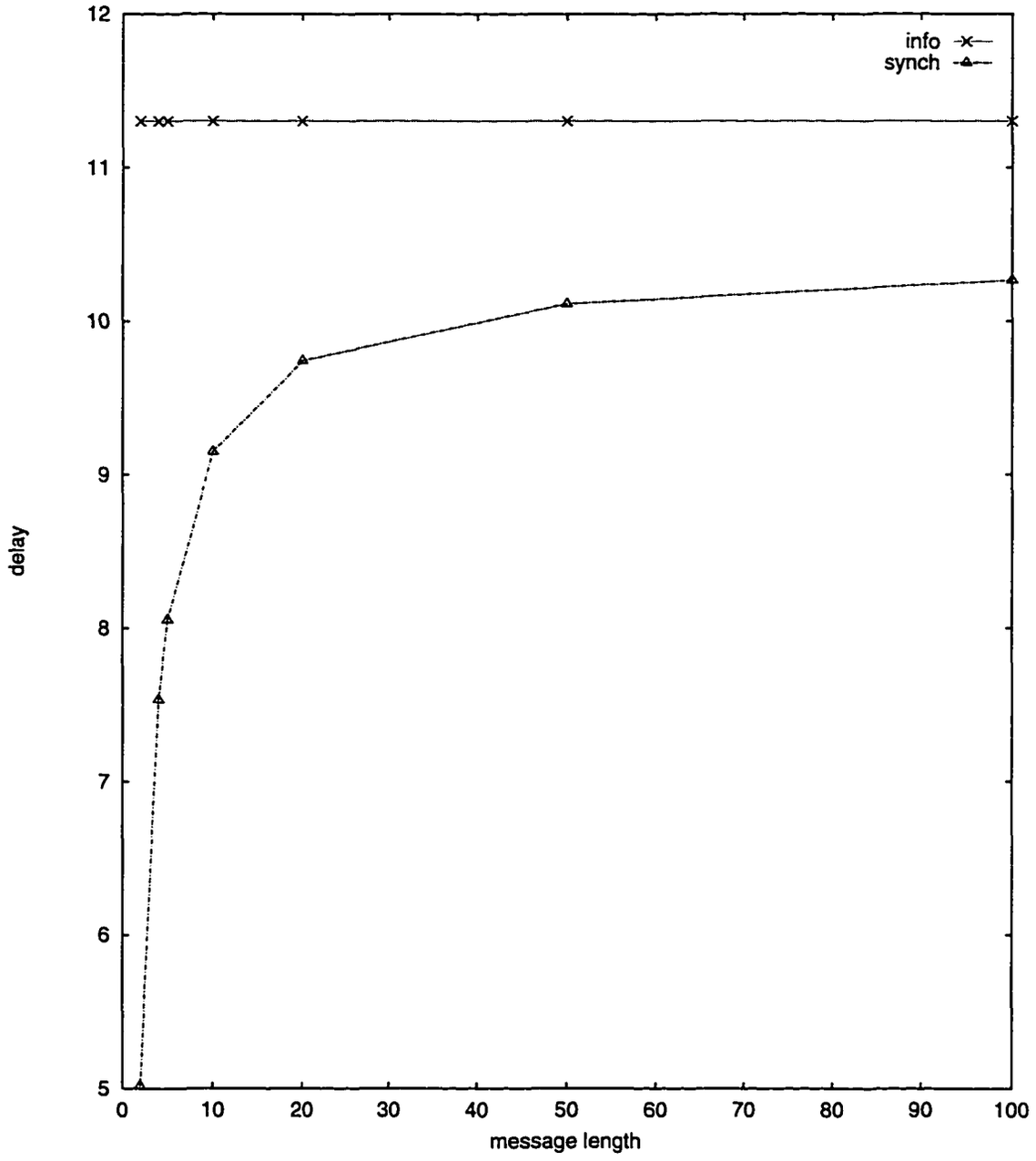

Figure 4.8: Expected Message Delay versus Message Length

then the overhead due to the additional information becomes less relevant.

To summarize, the average message delay for our implementation is superior to the average message delay for the waitfor implementation. Our occasional synchronization performs a natural form of flow control which prevents large transmission queue build-ups. However, our occasional synchronization also decreases the effective send rate. Our implementation is not preferable, from a performance standpoint, when the batch-size becomes very small. Our implementation is most beneficial when the batch-size is large and the average message size is small.

4.6 Chapter Summary

In this chapter we presented a second example of how our *delivered* and *delivered_all* constructs can be used in practice. We presented an implementation of flush channels based on our constructs. Flush channels can be an extremely useful construct for applications which require only a partial receipt order on messages. Previously suggested implementations of flush channels incorporate flush channels as a part of the underlying communication subsystem. Today, it is not realistic to expect the underlying communication system to support flush channels. Building flush channels into the system is inflexible and imposes an overhead on the system as a whole. The implementation of flush channels presented in this chapter provides flush channels at the user level; it allows applications to benefit from the relaxed ordering constraints and greater efficiency offered by flush channels without requiring kernel-level support for the construct. Providing flush channels at the user level is much more flexible and allows applications to use the construct selectively. An extensive performance study comparing our implementation to the waitfor technique was also presented in this chapter. We saw that our implementation was most advantageous, from

a performance standpoint, when the number of flush messages is small compared to the number of ordinary messages. Our implementation also compares more favorably for short messages.

Chapter 5

Transport Layer Vector Time

In earlier chapters we developed the formal tools needed to reason about transport layer information and illustrated how transport layer information could be used to provide alternative implementations of message ordering protocols. When examining the difference between application and transport layer information, it is also important to consider the impact on vector time. Vector time has emerged as a useful technique for solving problems in distributed systems. Although vector time is likely to be maintained by supporting code positioned in kernel space, vector time is traditionally viewed with respect to the application layer. In this chapter we consider vector time as perceived at the transport layer. We establish existing relationships between transport layer and application layer vector time. We show that transport layer vector time provides a more up-to-date view of the current system state, and therefore can improve performance for algorithms on which concurrency has an adverse effect. The possibility of updating vector time for acknowledgment messages is also considered. A new distributed termination detection algorithm based on this feature is presented.

5.1 Introduction and Motivation

In a sequential program, the events of that program are totally ordered by their occurrence in physical time. In an asynchronous distributed system, however, no common physical time line exists. As mentioned earlier, the events of a distributed system are only partially ordered based on causality[53]. Since no temporal order exists on the events in the system, physical clocks cannot be used to order the events. Vector time was introduced as a means of capturing the causal relationships present in the system[57, 34]. The relationships between the vector times of events in the system are isomorphic to the causal relationships between the events. Vector time has become a standard construct used for solving many problems in distributed systems. For example, vector time is used to provide causally consistent message passing in ISIS[14, 15], in algorithms for global predicate detection[27, 35, 36, 38], rollback recovery[75, 84], and deadlock detection[74], and in tools for distributed debugging[9, 31].

In the literature, vector time is traditionally considered to be present at the application layer. The vector time of a communication event corresponds to the vector time of the event occurrence at the application layer. However, vector time will most likely be maintained by supporting code positioned in the kernel space of a process. This dissertation is concerned with the distinction between information available at the application level of a process and at the kernel level of a process. As we have seen, the transport layer of a process which provides end-to-end communication, has often access to information not available at the application layer. It is important to examine the impact of this information on vector time. In this chapter we consider differences and relationships between application layer and transport layer vector time. Examining causality as present at the two layers is an important step towards understanding the differences between the two layers and towards

understanding how the information available at the transport layer can be used.

The causal relationships between communication events at the transport layer is not always consistent with the causal relationships between the events at the application layer. It is therefore crucial to make a distinction between vector time at the two layers. In this chapter we take a detailed look at communication to establish existing relationships between events at the two layers. We find that transport layer vector time gives a more up-to-date view of the "current causal time" in the system than application layer vector time. In addition, any causal relationships between two events in different processes present at the application layer are preserved at the transport layer. The impact of using transport layer vector time as opposed to application layer vector time for some sample applications from the literature is also considered. Some general guidelines for when transport layer vector time may be appropriate are given. We show that transport layer vector time can be beneficial for applications that gain from a decrease in concurrency. Another advantage of using transport layer vector time is that we can allow vector time to be updated for acknowledgment messages, not seen at the application layer, as well as for regular messages. Assuming a system that uses reliable message passing based on a positive acknowledgment scheme, we use this feature to derive a new distributed termination detection algorithm. The algorithm is intuitive and simple when using transport layer vector time but would not be feasible based on application layer vector time.

5.2 Causality and Vector Time

As mentioned above, one important characteristic of a distributed system is the lack of a global clock. Due to the lack of a common physical time line, no temporal order exists on the events of a distributed system. Instead, the events of the system are only partially

ordered based on causality[53]. Recall from Chapter 3 that the causality relation is the smallest relation satisfying the following three conditions:

1. If e and e' are events in the same process and e occurs before e', then $e \rightarrow e'$.

2. If e is the sending event of a message by one process and e' is the corresponding receive event in another process, then $e \rightarrow e'$.

3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Recall that two events, e and e', are said to be concurrent, $e \parallel e'$, if they are not ordered by causality. The send event for message m is denoted by $s(m)$ and the receive event for message m is denoted by $r(m)$.

Vector time was introduced by Mattern[57], and independently by Fidge[34], as a means for capturing the causal relationships present in the system. Vector time is an extension of Lamport's logical clocks[53]. In an N-process system, $\{P_0, P_1, \ldots, P_{N-1}\}$, each process, P_i, maintains a vector, V_i, of N logical clocks or counters, indexed 0 through $N-1$. The value of component j of P_i's vector time, V_i^j, represents the best approximation process P_i can make about the current "time" in process P_j. In general, only component i of P_i's vector time is completely accurate. Each time a process sends a message, it appends its vector time to the message, conveying its current view of time in the system to the receiving process. Initially, $V_i^j = 0$ for all i,j. The vector times are then maintained according to the following rules:

1. If event e occurs in P_i, then $V_i^i = V_i^i + 1$.

2. If event e is a send event in P_i and e' is the corresponding receive event in P_j, then $V_j = sup(V_i, V_j)$, where $sup(V_i, V_j) = \left(max(V_i^0, V_j^0), \ldots, max(V_i^{N-1}, V_j^{N-1})\right)$.

For event e, the vector time of e, denoted $V(e)$, is the vector time in the process where e occurred. Thus, if e is an event on P_i, then $V(e)$ is the value of V_i at the occurrence of e. The vector time of event e is assigned after the vector time, V_i, has been properly updated. The events in the system include, but are not limited to, all send and receive events. Mattern defines the following relationships between any two vector times W and U:

1. $W \leq U$ iff $\forall k : W^k \leq U^k$

2. $W < U$ iff $(W \leq U) \wedge (W \neq U)$

3. $W \parallel U$ iff $(W \not< U) \wedge (U \not< W)$

The relationship between the vector timestamps of two events is isomorphic to the causal relationship between the events. For events e and e', $e \rightarrow e'$ if and only if $V(e) < V(e')$ and $e \parallel e'$ if and only if $V(e) \parallel V(e')$.

5.3 Relationships

Communication events result in activity at multiple layers of a system. The definition of vector time in the previous section does not concern itself with where in the system send and receive events occur. The traditional view in the literature is to consider vector time with respect to communication events occurring in the application. However, it is also important to consider vector time as associated with the communication events as they occur at the transport layer. It is necessary to make a distinction between vector time at the two levels since the causal relationships between send and receive events may not be consistent between the two levels. In this section, we examine relationships existing between application and transport layer vector time. We assume that vector time is only updated

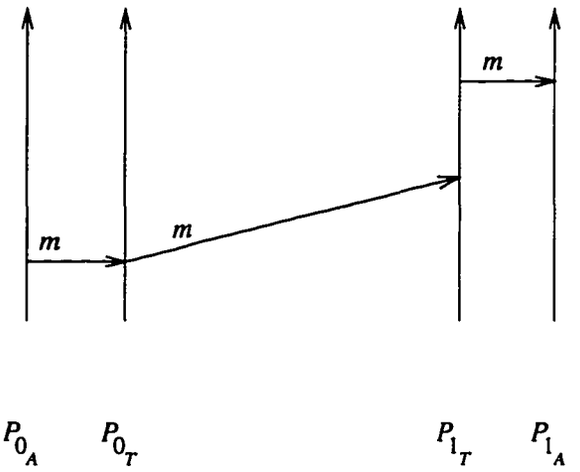

Figure 5.1: Communication Process

due to communication, since the computational events of an application may not be seen at the transport layer. Similarly, we assume for the moment that possible acknowledgment messages, present only at the transport layer, do not affect vector time. Recall from earlier chapters that we assume a receiving transport layer passes messages to the application in the same order they arrive at the receiving transport layer.

To establish the relationships existing between application and transport layer vector time, we will first examine the communication process. Communication between two application processes involves several steps. The sending application process sends the message to its transport layer by executing a *send* system call. The transport layer is then responsible for delivering the message to the transport layer of the receiver. The receiving application process obtains the message from its transport layer through a *receive* system call. This communication is illustrated in Figure 5.1. In the figure, P_{i_A} denotes the application process in process P_i and P_{i_T} denotes the transport layer. The causal chain for sending a message m in this detailed communication view is expressed by the following

Communication Property:

Communication Property: $s_A(m) \rightarrow s_T(m) \rightarrow r_T(m) \rightarrow r_A(m)$

where the subscripts A and T denote communication events as perceived by the application process and transport layer respectively. Note that the Communication Property is consistent with our formal system model presented in Chapter 2. It is a straightforward extension of the Network Axioms expressed in causal terms.

Vector time could be easily modified to capture the causal relationships expressed by the Communication Property. We would simply include two components for each process in the vector time. For example, we could let V^{2i} denote the time at the application layer in process P_i and V^{2i+1} denote the time at the transport layer in process P_i. Although this extension to vector time would capture the application layer's as well as the transport layer's view of the causal relationships in the computation, it is not very practical. It requires two components for each process, which doubles the size of the vector timestamp included in each message. Instead, we examine the relationships between causality as perceived by the application layer and the transport layer. We can then determine which form of vector time is more appropriate for a specific application.

To see why the causal relationships between events may differ between the two layers, consider the computation shown in Figure 5.2. We can see that message m_1 is delivered to the transport layer of P_1 *before* message m_2 is sent. However, message m_1 is buffered by the transport layer in the receiver and it is not received by the application process until *after* message m_2 has been sent. Assuming the system uses blocking sends, a send system call will not return until the transport layer has space in its send buffer. Thus, we can view send events as being atomic across the layers with respect to incoming messages and other send events. However, this is not true for receive events. A message delivered at

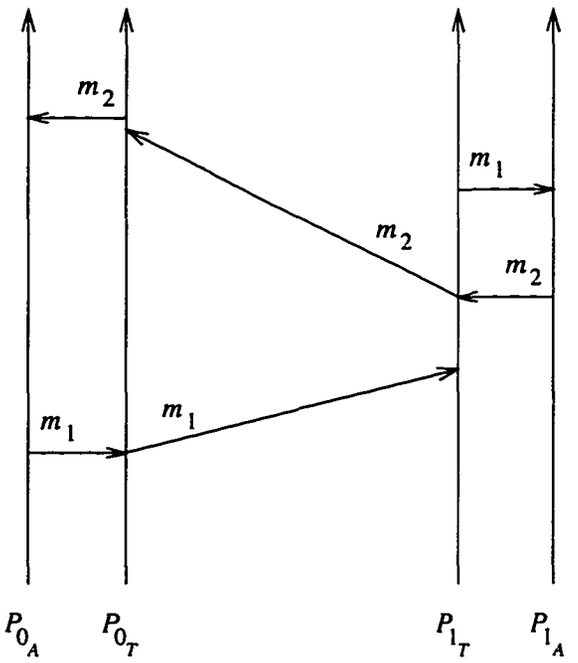

Figure 5.2: Sample Computation

the transport layer of a process may be buffered indefinitely until the application process, for which the message is destined, executes a *receive* system call. Due to this buffering, the causal relationship between events can differ between the application and the transport layer. Abstracting the application layer's view of the computation in Figure 5.2 we obtain the space-time diagram shown in Figure 5.3. Abstracting the transport layer's view produces a different diagram as shown in Figure 5.4. We can see from Figures 5.3 and 5.4 that the causal relationships present between send and receive events in the computation depend on whether the computation is viewed from the application layer or the transport layer. As reflected by the displayed vector times, from the application's view $a \parallel b$ and $b \rightarrow c$. From the transport layer's view, on the other hand, $a \rightarrow b$ and $c \rightarrow b$.

As illustrated above, there are differences between the causal relationships of events at

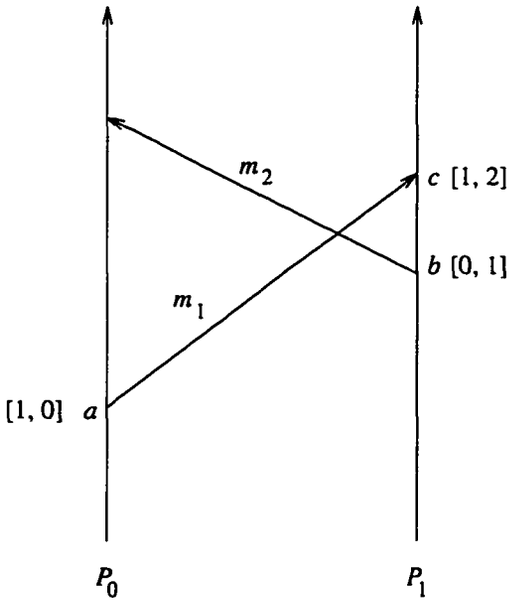

Figure 5.3: Application Layer View

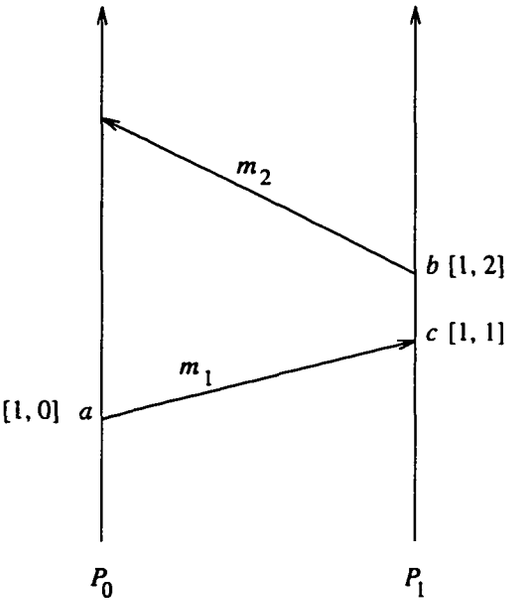

Figure 5.4: Transport Layer View

the application layer and transport layer. However, as we will see there are relationships between vector time at the two levels which can be shown to hold. For a send or receive event, e, we will use e_A to represent the occurrence of e at the application layer and e_T to represent the occurrence of e at the transport layer; we will use $V_A(e)$ to denote the vector time of the send or receive with respect to the application layer and $V_T(e)$ to denote the vector time of the send or receive with respect to the transport layer. We will talk about send and receive events for a message as single events viewed from either the transport layer or the application layer. However, we must keep in mind that a receive event from the viewpoint of the transport layer occurs when the message is delivered to the transport layer. This may happen long before the event is observed at the application layer. As mentioned above, we view sends as atomic across the layers. As long as the system uses blocking sends

this is reasonable and easy to implement. Assuming atomic sends allows us to utilize the Atomic Send Property described below:

Atomic Send Property: For any event e, $e \neq s_A(m) \wedge e \neq s_T(m)$,

1. $s_A(m) \rightarrow e$ if and only if $s_T(m) \rightarrow e$
2. $e \rightarrow s_A(m)$ if and only if $e \rightarrow s_T(m)$

For process P_i, vector time as maintained at the application layer is denoted by V_{i_A} and vector time as maintained at the transport layer is denoted by V_{i_T}. When convenient and no confusion exists we will drop the first subscript and simply use V_A to denote vector time with respect to the application layer and V_T to denote vector time with respect to the transport layer. Vector time is maintained according to the same rules, described in Section 5.2, both at the application layer and at the transport layer.

Incoming messages provide information about the progress of the computation. When an incoming message is buffered, this information is temporarily stalled at the transport layer. In a sense, the transport layer is more well-informed than the application layer. This fact is expressed by Property 1:

Property 1 $V_A \leq V_T$

Proof: Recall that we assume that only communication events update vector time. Send events are considered to be atomic across the layers, which means that vector time is updated simultaneously at the two layers for a send event. For receive events, we know from the Communication Property that $r_T(m) \rightarrow r_A(m)$. For a receive event, vector time is updated at the transport layer prior to the event occurrence at the application layer. Thus, the transport layer always has a more accurate or equal view of the "current time" in the system. ∎

When the application layer receives a buffered message it "catches up" with the knowledge available at the transport layer. As expressed by Property 2, there is a straightforward relationship between the local time perceived at the two layers and the number of messages in the receive buffer.

Property 2 $V^i_{i_A} = V^i_{i_T}$ *– the number of messages in the receive buffers at* P_i.

Proof: Send events are viewed as atomic across the layers so the same number of sends have been executed at both layers. The number of receive events performed by the application equals the number of messages received by the transport layer minus the messages which are still buffered. Hence, the relationship described above. ∎

Note that when there are no messages in the receive buffers at the transport layer the perception of local time in a process is "synchronized" between the two layers. Even though the local component of vector time is synchronized between the layers when there are no outstanding messages in the receive buffers, this may not be true for the other components. Due to the transitive effect of buffering, the transport layer may still be more well-informed about the current time in the system than the application layer even when there are no messages in the receive buffers. This situation is illustrated in Figure 5.5, which displays vector time as perceived both at the application layer and at the transport layer. Note that the vector timestamps appended to the messages in the computation depend on what form of vector time is used. When application layer vector time is used, the timestamp appended to message m_2 is $[0, 1, 0]$ and when transport layer vector time is used the timestamp is $[0, 2, 1]$. In the figure, after message m_2 has been received by the application there are no messages in the receive buffers at process P_0. Yet at this point the vector time at the application layer is [1,1,0] and the vector time at the transport layer is [1,2,1]. The difference in vector time between the two layers is due to the fact that message m_1 has been delivered

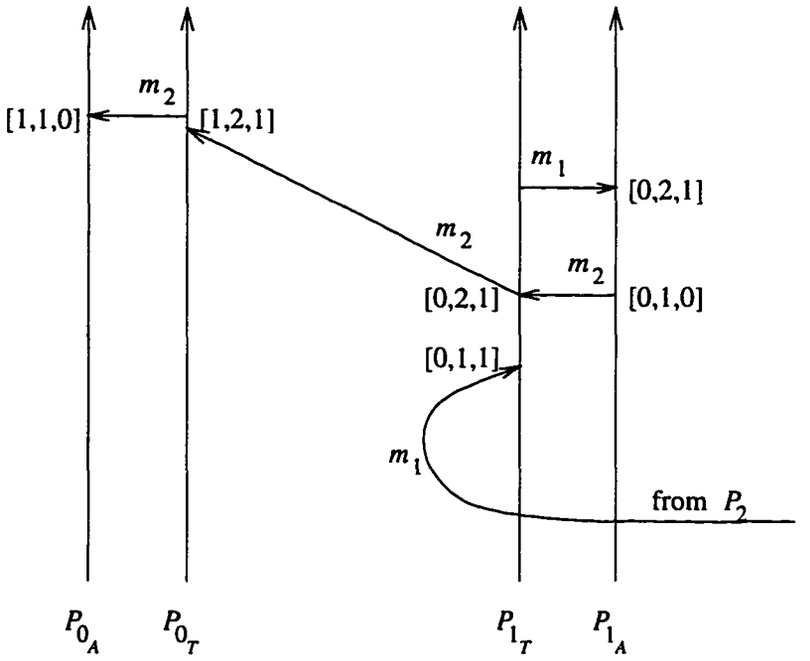

Figure 5.5: Transitive Buffering Effect

but not yet received at process P_1 when message m_2 is sent. The difference in vector times at process P_0 is caused by the buffering of message m_1 in process P_1.

Several relationships can be established between the causal relationship of two events as perceived by the two layers. For two events e and e' in the same process, the following properties hold:

Property 3 *Let e and e' be send events in the same process, then $V_A(e) < V_A(e')$ if and only if $V_T(e) < V_T(e')$.*

Property 4 *Let e and e' be receive events in the same process, then $V_A(e) < V_A(e')$ if and only if $V_T(e) < V_T(e')$.*

Property 5 *Let e be a send event and e' be a receive event in the same process, then* $V_T(e) < V_T(e') \Rightarrow V_A(e) < V_A(e')$.

Property 6 *Let e be a receive event and e' be a send event in the same process, then* $V_A(e) < V_A(e') \Rightarrow V_T(e) < V_T(e')$.

Proof: Since we assume send events to be atomic across the layers successive send events in a process will occur in the same order at both layers. Property 3 follows trivially from this fact. Similarly, messages are assumed to be passed to the application in the same order they were delivered at the transport layer. Hence, messages are received in the same order at both layers and Property 4 follows trivially. To show Property 5, let e be the send event for message m and e' be the receive event of message m'. Assume that $V_T(e) < V_T(e')$ which means that $s_T(m) \rightarrow r_T(m')$. From the Communication Property we know that $r_T(m') \rightarrow r_A(m')$. Combining the facts we have $s_T(m) \rightarrow r_A(m')$. From the Atomic Send Property we then have $s_A(m) \rightarrow r_A(m')$ which implies that $V_A(e) < V_A(e')$. This establishes Property 5. Similarly for Property 6, let e be the receive event of message m and e' be the send event of message m'. Assume that $r_A(m) \rightarrow s_A(m')$. From the Communication Property we have $r_T(m) \rightarrow r_A(m) \rightarrow s_A(m')$. The Atomic Send Property then gives $r_T(m) \rightarrow s_T(m')$. This implies that $V_T(e) < V_T(e')$ and establishes Property 6. ∎

For events in different processes, the relationships between the application layer and transport layer view is straightforward. Any causal relationships present at the application layer are preserved at the transport layer. This is expressed by Property 7:

Property 7 *Let e and e' be events in different processes, then* $V_A(e) < V_A(e') \Rightarrow V_T(e) < V_T(e')$.

Proof: Assume $V_A(e) < V_A(e')$ and e and e' are events on different processes. Then there

must exist a message or a chain of messages creating a causal link between e and e'. Without loss of generality, assume there exists a single message m. Then we have $e_A \rightarrow s_A(m) \rightarrow r_A(m) \rightarrow e'_A$. By Properties 3 and 6 we have $e_T \rightarrow s_T(m)$, from the Communication Property we have $s_T(m) \rightarrow r_T(m)$, and from Properties 4 and 6 we have $r_T(m) \rightarrow e'_T$. Combining the three facts we conclude $e_T \rightarrow e'_T$, and thus $V_T(e) < V_T(e')$. If there are several messages on the causal chain the same argument can be repeated for each message. The case where $e = s(m)$ or $e' = r(m)$ only simplifies the argument. Hence, Property 7 is established. ∎

Property 7 applies to two events in distinct processes. However, it can of course also be applied to two events in the same process if they are causally linked through a third event e'', $e \rightarrow e'' \rightarrow e'$, where e'' is an event in a different process. It follows from the contrapositive of Property 7 that $V_T(e) < V_T(e') \Rightarrow (V_A(e) < V_A(e')) \vee (V_A(e) \parallel V_A(e'))$, for e and e' events in different processes. This is consistent with Property 1 which tells us that the transport layer has a more up-to-date view of the current time in the system. Since the transport layer is better informed than the application layer, it captures some causal relationships which are not visible at the application layer. As we will see in the next section this can be very useful for some applications.

5.4 Impact on previous work

The differences and relationships present between application and transport layer vector time were described in the previous section. In this section we will examine the impact of using transport layer vector time instead of application layer vector time for two sample applications. To illustrate under what conditions a particular type of vector time is appropriate, we will consider two well-known applications of vector time: Cooper and Marzullo's

algorithms for global predicate detection[27] and the implementation of causally consistent multicast in ISIS[14].

5.4.1 Global Predicate Evaluation

Let us first consider the work on global predicate detection done by Cooper and Marzullo[27]. Their algorithms for detecting *possibly* Φ and *definitely* Φ, for global predicate Φ, are based on vector time. Each process in the system reports communication events and other relevant changes in their local state to a monitoring process. Based on the vector time of the reported events, the monitoring process can then construct a lattice of global states which is consistent with the observed execution of the system. A node in the lattice corresponds to a consistent cut[57] of the computation; the global state *could have occurred*, in real time, during the execution without violating the observed causal dependencies. A point in the lattice, $\bar{x} = (x_0, x_1, \ldots, x_{n-1})$, represents a global state where process p_i has executed x_i relevant events. The level of a point, $\bar{x}$, is defined to be the sum of the components of the vector, $x_0+x_1+\cdots+x_{n-1}$. A path through the lattice where the level of each successive point in the path increases by one then forms a sequence of consistent global states representing a possible execution path for the observed computation. An execution of a two-process system and the corresponding lattice are shown in Figures 5.6 and 5.7, respectively. A point $S_{i,j}$ in the lattice denotes the point $\bar{x} = (i, j)$. One possible execution path for the observed execution is given by the global state sequence

$$S_{0,0}; S_{1,0}; S_{2,0}; S_{2,1}; S_{3,1}; S_{3,2}; S_{3,3}; S_{3,4}; S_{4,4}$$

All states in the lattice represent global states reachable in a possible execution of the

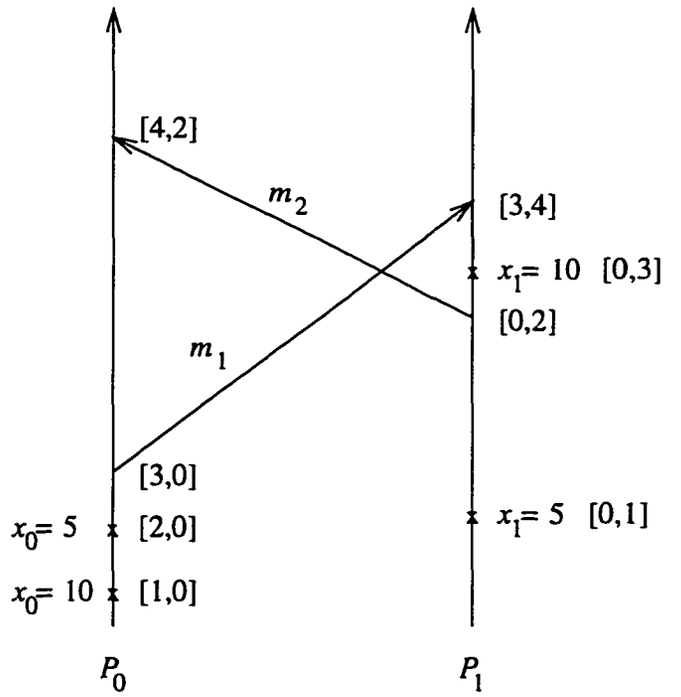

Figure 5.6: Application Layer View

observed computation. Hence, the *possibly* Φ predicate is satisfied if there is a point in the lattice at which Φ is satisfied. When *possibly* Φ is true, all we know is that Φ is satisfied in some consistent cut of the computation. No definite knowledge is obtained. The predicate, Φ, may or may not have been satisfied in a real-time global state of the computation. On the contrary, when *possibly* Φ is false we know with certainty that Φ was never satisfied during the execution; the observed computation could not exhibit a consistent cut in which Φ was true. The algorithm to evaluate *possibly* Φ constructs the lattice level-by-level searching for a point at which the predicate is satisfied. Only the global states of the current level and local states that may be part of higher-level global states are maintained. For *definitely* Φ to be satisfied every possible execution of the observed computation must pass through a state

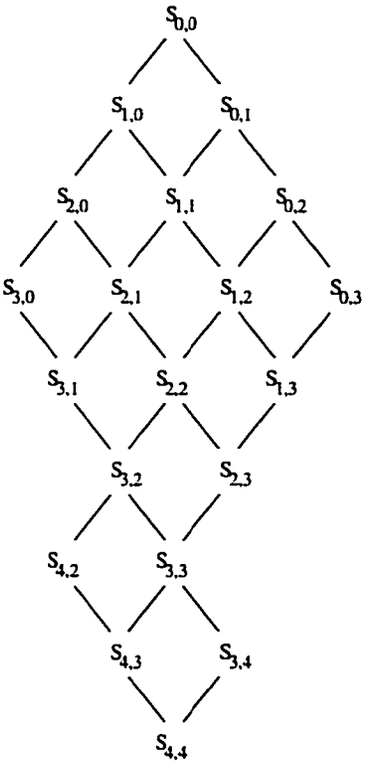

Figure 5.7: Corresponding Lattice

at which Φ is satisfied. Hence, *definitely* Φ is satisfied if all possible paths through the lattice pass through a point at which Φ is satisfied. When *definitely* Φ is true we know with certainty that the predicate was satisfied at some real-time instant of the observed computation; a consistent cut in which Φ is true must have been realized during the execution. Little information is obtained when *definitely* Φ is false since the predicate may or may not have been satisfied; a consistent cut in which Φ is true may still exist and could have been realized during the execution. The algorithm to determine *definitely* Φ builds the lattice level-by-level, but for each level it only maintains global states at which Φ is not satisfied. If the algorithm reaches a point at which no state can be constructed for the next level,

then it concludes *definitely* Φ to be true.

Let us now examine the impact of the type of vector time used on the algorithms. The execution shown in Figure 5.6 is the execution from Figure 5.3 augmented with some additional events which modify the local variables x_0 and x_1. Again, let the execution shown in Figure 5.6 represent the application's view of the computation. The lattice corresponding to the view of the application is shown in Figure 5.7. Consider the evaluation of the global predicates *possibly* $(x_0 = 10 \wedge x_1 = 10)$ and *definitely* $(x_0 = 5 \wedge x_1 = 5)$. Since *possibly* $(x_0 = 10 \wedge x_1 = 10)$ is satisfied as long as the predicate evaluates to true at any point in the lattice, we can see that the algorithm will return a positive answer. The predicate $(x_0 = 10 \wedge x_1 = 10)$ is satisfied at lattice point $S_{1,3}$. Next, let us consider the evaluation of *definitely* $(x_0 = 5 \wedge x_1 = 5)$. For *definitely* $(x_0 = 5 \wedge x_1 = 5)$ to evaluate to true, every possible execution path must pass through a lattice point at which the predicate evaluates to true. Consider the path

$$S_{0,0}; S_{1,0}; S_{1,1}; S_{1,2}; S_{1,3}; S_{2,3}; S_{3,3}; S_{3,4}; S_{4,4}$$

We can see that $(x_0 = 5 \wedge x_1 = 5)$ is not satisfied at any of the points along the path. Thus, *definitely* $(x_0 = 5 \wedge x_1 = 5)$ will be false. Little or no information about the observed execution was gained by evaluating *possibly* $(x_0 = 10 \wedge x_1 = 10)$ and *definitely* $(x_0 = 5 \wedge x_1 = 5)$ based on application layer vector time.

The computation as viewed by the transport layer is shown in Figure 5.8. This is the computation from Figure 5.4, augmented with the extra local events. The corresponding lattice is shown in Figure 5.9. Since relevant changes must be reported to the monitoring process such changes will be visible at the transport layer as well as the application

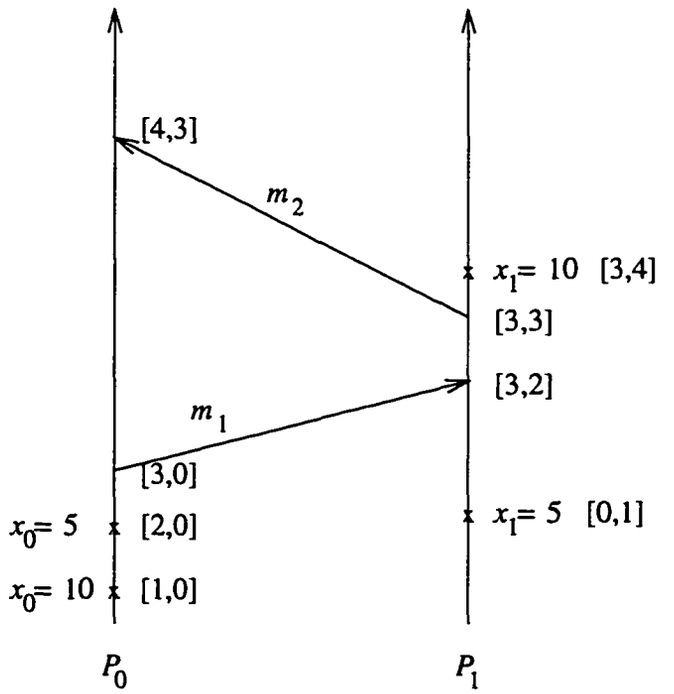

Figure 5.8: Transport Layer view

layer. Although the control messages are not shown in Figures 5.6 and 5.8, we can view the updates of vector time as occurring when the control messages are sent rather than when the modifications to the local variables occur. We can see that, if we use transport layer vector time, then *possibly* $(x_0 = 10 \wedge x_1 = 10)$ will be evaluated to false. The point $S_{1,3}$ is no longer in the lattice since it no longer represents a consistent global state. The predicate $(x_0 = 10 \wedge x_1 = 10)$ is not satisfied at any point in the lattice. Thus, the result of evaluating *possibly* $(x_0 = 10 \wedge x_1 = 10)$ is different when transport layer vector time is used. Similarly, *definitely* $(x_0 = 5 \wedge x_1 = 5)$ will be evaluated to true. The predicate $(x_0 = 5 \wedge x_1 = 5)$ is satisfied at lattice point $S_{3,1}$. Since $S_{3,1}$ is the only point in the lattice at level four, all possible paths through the lattice must pass through this point.

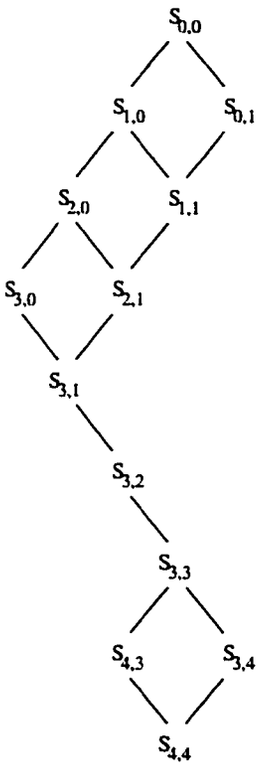

Figure 5.9: Corresponding Lattice

Hence, *definitely* $(x_0 = 5 \wedge x_1 = 5)$ is satisfied. Evaluation of the predicates based on transport layer vector time provides useful information about the observed execution. We know with certainty that predicate $(x_0 = 5 \wedge x_1 = 5)$ was true at some point during the observed computation and that predicate $(x_0 = 10 \wedge x_1 = 10)$ was never satisfied.

The transport layer has a more up-to-date view of the global state of the system. At the transport layer we have sufficient information to deduce that predicate $(x_0 = 10 \wedge x_1 = 10)$ was not satisfied and that predicate $(x_0 = 5 \wedge x_1 = 5)$ was satisfied at an instant of the observed computation. The information available at the application layer is not sufficient to draw such conclusions. The more informed view at the transport layer decreases the

number of concurrent events in the system. The number of global states in the lattice are reduced and more informed conclusions can be drawn. The global states that are pruned from the application layer lattice are global states which could not have been realized during the execution; based on the causal information available at the transport layer they do no longer correspond to consistent cuts of the computation. It follows from Theorem 7 that the set of global states that constitute the transport layer lattice for an observed computation is a subset of the global states that constitute the corresponding application layer lattice. This means that, whenever *possibly* Φ evaluates to false based on application layer vector time, it will also evaluate to false based on transport layer vector time. Similarly, whenever *definitely* Φ evaluates to true at the application layer, it will also evaluate to true at the transport layer. Hence, the results obtained when transport layer vector time is used are always at least as informative as the results obtained when application layer vector time is used. As our example above showed, the reverse is not true. Evaluating the predicates based on transport layer vector time produced more informative results than evaluating the predicates based on application layer vector time. Since the number of states in the lattice is reduced the use of transport layer vector time is also more computationally efficient. The complexity of both algorithms is linear in the number of possible global states. Thus, we see that, for the algorithms presented by Cooper and Marzullo, transport layer vector time allows more informative conclusions and is more efficient.

5.4.2 Causally Consistent Multicast in ISIS

As a second example let us consider the use of vector time to implement causally consistent multicast communication in ISIS[14, 15]. All communication in ISIS is in the form of multicasts within process groups. The ISIS system provides causally consistent multicast

through its CBCAST protocol[14]. Multicast communication is causally consistent if for all processes P_i in the process group:

$$s(m) \rightarrow s(m') \Rightarrow r_i(m) \rightarrow r_i(m')^1$$

Causal consistency is ensured in ISIS by buffering an incoming message until all messages which causally precede it have been received by the application. Whether an incoming message must be buffered or not can be deduced from a vector timestamp generated at the time of its transmission and piggybacked on the message. When message m, sent by process P_i and timestamped with $V(m)$, arrives at process $P_j, i \neq j$, it is buffered until:

$$\forall k : 0 \cdots N-1 \begin{cases} V^k(m) = V_j^k + 1 & \text{if } k = i \\ V^k(m) \leq V_j^k & \text{otherwise} \end{cases}$$

Messages sent by process P_j itself are never buffered.

To illustrate the difference in using application layer and transport layer vector time in the CBCAST implementation provided by ISIS, let us consider the message passing shown in Figures 5.10 and 5.11. This is again the computation from Figures 5.3 and 5.4, however this time messages m_1 and m_2 are also multicast to additional processes. From the viewpoint of the application, $V(m_1) = [1,0]$ and $V(m_2) = [0,1]$, hence messages m_1 and m_2 are concurrent. Using application layer vector time the CBCAST protocol will allow a process to receive messages m_1 and m_2 in any order. Using transport layer vector time instead, $V(m_1) = [1,0]$ and $V(m_2) = [1,2]$, hence $s(m_1) \rightarrow s(m_2)$. The CBCAST protocol would require that message m_1 be received before message m_2. To enforce this requirement,

[1]Note that our use of *receive* and *deliver* is reversed from the notation used by Birman et al. in their presentation of the CBCAST protocol.

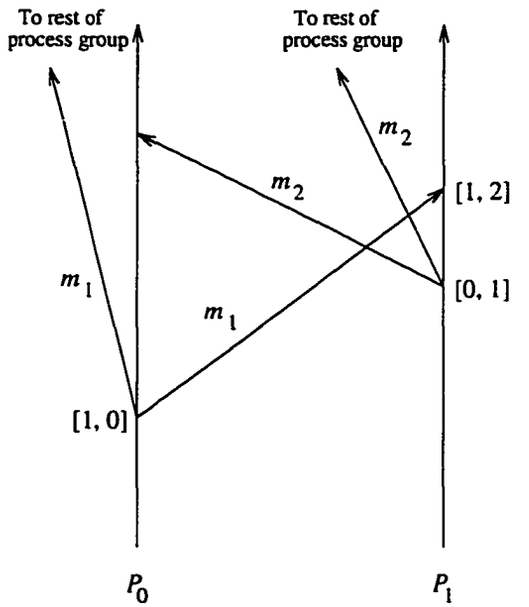

Figure 5.10: Application Layer View

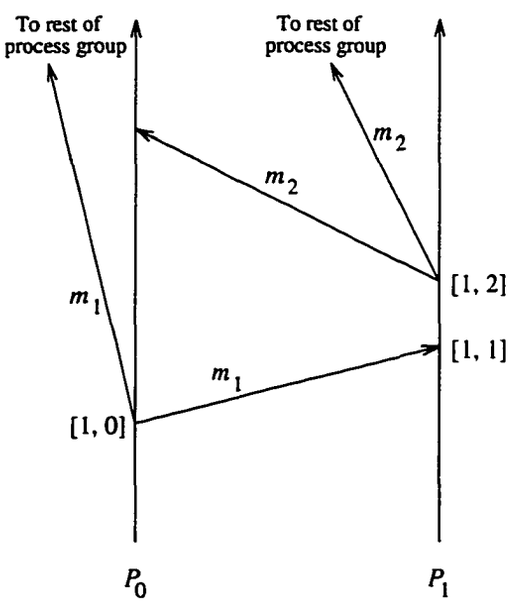

Figure 5.11: Transport Layer View

the CBCAST protocol would have to be modified to allow a process to buffer messages sent by itself. In our example, process P_1 would have to buffer message m_2 until message m_1 has been received to avoid violating the causal consistency requirement.

The purpose of causally consistent multicast is to ensure that, if message m may have influenced the contents of message m' or even caused the sending of m', then m will be received before m' by all processes. Since message sends are triggered by the application, the sent message cannot be influenced by messages which have only been delivered and not yet received. Here we are interested in the causal relationship between messages at the application layer and thus the proper vector time to use is application layer vector time. Using transport layer vector time induces redundant ordering constraints on messages. It is also awkward for a process sometimes to be required to buffer messages sent by itself. Transport layer vector time is both less efficient and less intuitive to use for this application.

5.4.3 General Guidelines

For the applications considered above, transport layer vector time showed to be more appropriate in one case and application layer vector time was more appropriate in the other case. A question then naturally arises: what is the difference between the two applications? Are there structural differences between the two applications which cause transport layer vector time to work well for our first example but not for our second example? Examining these questions will help us identify the type of applications for which transport layer vector time may be useful.

We saw that transport layer vector time was more appropriate for Cooper and Marzullo's algorithms. In this application, vector time was used to provide a partial order on the events in the system, which were then used to construct possible global states. The more well-informed system view present at the transport layer reduced the number of concurrent events and as a result reduced the number of possible global states. Algorithms which benefit from a reduction in the number of concurrent events in the system are likely to benefit, both in terms of efficiency and in terms of accuracy, from the use of transport layer vector time. Algorithms, such as Cooper and Marzullo's, which attempt to construct a "global view" of the system fall into this category. For the implementation of causally consistent multicast in ISIS, application layer vector time was shown to be more appropriate. In this application, vector time was used to ensure that application-level causal constraints on message passing events where not violated. For this application the transport layer was, in a sense, too well-informed. Using transport layer vector time enforced causal constraints which were not relevant. This resulted in a loss of efficiency. Application layer vector time is more appropriate for algorithms which benefit from an increase in concurrency. Algorithms, such as the ISIS implementation, which enforce some minimal causal constraints, fall into this

category. It should also be noted that applications that depend on causal dependencies that are different at the application layer and at the transport layer may be forced to use the form of vector time which represents the relevant causal relationships.

5.5 Updates on ACKs

The previous section illustrated how transport layer vector time could improve performance for certain classes of algorithms by reducing the number of causally concurrent events in the system. Transport layer vector time is also more powerful than application layer vector time in that it can allow vector time to be updated for acknowledgment messages (ACKs) as well as for regular messages. In a system employing reliable message passing, the transport layer must ensure that all messages are correctly delivered to the transport layer at the receiving process. This is normally done through some positive acknowledgment algorithm, such as a sliding window protocol. When considering vector time as maintained at the transport layer, it is important to also consider the possibility of letting ACKs update vector time. Since the acknowledgment messages are only seen at the transport layer, this is not possible for application layer vector time. Acknowledgments provide additional structure to the computation and it is only natural to take the opportunity to convey this information through vector time. Allowing information to flow through the ACKs will give a more up-to-date view of the system state which will cut down on the number of concurrent events in the system. Decreasing the number of concurrent events in the system, through vector time updates on ACKs, can even further enhance the efficiency and accuracy of some algorithms. Consider the algorithms by Cooper and Marzullo discussed earlier on the computation shown in Figure 5.12. In the figure, acknowledgment messages are indicated by dashed lines. The lattice corresponding to a system using regular transport layer vector

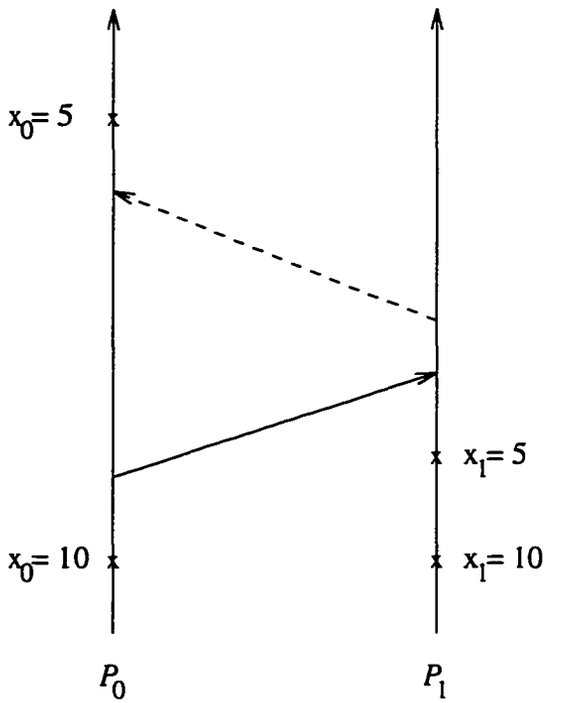

Figure 5.12: Sample Computation

time is shown in Figure 5.13 and the lattice corresponding to a system where vector time is updated on ACKs is shown in Figure 5.14. We can see that the lattice in Figure 5.14 allows more informative conclusions. From the lattice in Figure 5.14 we can conclude that *definitely* $(x_0 = 10 \wedge x_1 = 5)$ is true and that *possibly* $(x_0 = 5 \wedge x_1 = 10)$ is false, something which is not possible from the lattice in Figure 5.13. Allowing ACKs to update vector time also increases the efficiency of the algorithm as a result of decreasing the number of possible global states.

The acknowledgment messages carry information about the safe delivery of one or several messages. We must be aware that, when we use the information provided by the ACKs, we may rely on causal dependencies that are not represented by regular vector time. If

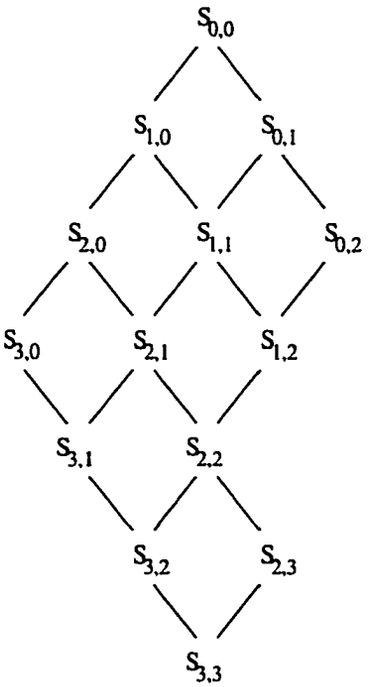

Figure 5.13: Corresponding Lattice with No Vector Time Update by ACKs

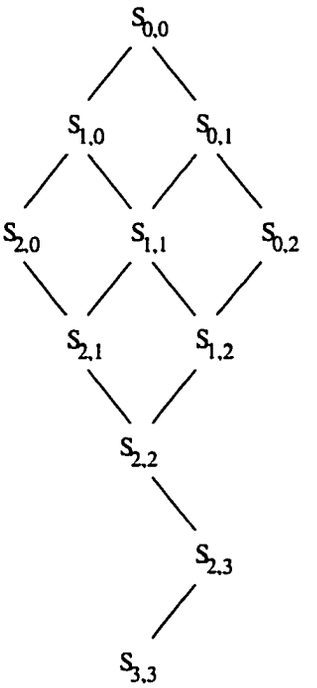

Figure 5.14: Corresponding Lattice when ACKs Update Vector Time

we use causal links introduced by ACKs we must also provide an accurate representation of causality. Any algorithm based on vector time which utilizes causal links introduced by ACKs will also require vector time to be updated by ACKs. In the next section we will derive a distributed termination detection algorithm that relies on this fact.

Considering the possibility to update vector time on ACKs also poses a conceptual question. If vector time is mapping causality exactly [57], then how can there be an option on when to update vector time? There should only be one causality relation present in the system. One approach to try and resolve the contradiction would be to view vector time as a means of exactly capturing the pertinent causality in the system. Changing our notion of which causal links in the system are relevant would then allow us to change the occasions at

which vector time must be updated. Note that the same conceptual problem does not arise when considering the distinction between application and transport layer vector time. The delivery of a message to the transport layer and the receipt of the message at the application layer are two time-distinct events. They may have different causal relationships to other events in the system as reflected by the possible difference in the vector timestamp of the events.

5.6 Termination Detection Based on Vector Time

Just as vector time is assumed to exist at the application level, the control algorithm handling termination detection is traditionally viewed to exist at the application layer, "above" the basic computation. However, in practice the control algorithm is more likely to be positioned in kernel space rather than as another user process. In this dissertation we are interested in the improved use of low-level information. As mentioned earlier, a user process sending a message in a system using reliable asynchronous message passing only knows that this message will eventually reach its destination. The transport layer is responsible for ensuring the reliability of the message. The transport layer support code must keep a copy of the message in its buffers until it is notified that the message has been properly delivered. Hence, at the kernel level we can determine whether all messages sent by a process have been safely delivered or not. In Chapter 2, we derived constructs for passing this information to the user level. Subsequent chapters demonstrated how the constructs could be used. The distributed termination detection algorithm derived in this chapter will illustrate how information about delivery of messages can be used directly by control algorithms that reside in kernel space. If it is desired to run the termination detection algorithm as a user process, then the primitives presented in Chapter 2 can be used to propagate information

on delivery of messages to the algorithm. Our protocol will use transport layer vector time. We will assume that the protocol employed by the networking software uses positive acknowledgments. This will allow us to update vector time for acknowledgment messages as well as for regular messages. Since we use our knowledge about properly delivered messages, it is necessary to update vector time for acknowledgment messages to represent the pertinent causality present in the system.

5.6.1 The Termination Detection Problem

Before proceeding, we define the termination detection problem. As before, we are considering a computation consisting of a set of N processes. The processes are communicating only through reliable asynchronous message passing. The termination detection problem can be described as follows [101]:

- A process must be in one of two states, *active* or *passive*.
- Only active processes are allowed to send messages.
- An active process may spontaneously become passive at any time.
- A passive process only becomes active upon the receipt of a message.
- The computation is terminated when all processes are concurrently passive and there are no messages in transit.

A correct solution to the problem must ensure the following safety and liveness conditions:

Safety: If termination is detected, then the computation is terminated.

Liveness: If the computation is terminated, then termination will eventually be detected.

5.6.2 The Algorithm

Having defined the termination detection problem we are now ready to present our solution. Our algorithm uses a centralized server which will maintain information allowing it to conclude termination of the system. The algorithm is extremely simple:

> *The centralized server maintains an array of N vector timestamps, one timestamp for each process in the system. Each process is required to send a message to the server whenever it changes its state from active to passive and it has no outstanding messages. This message updates vector time in the usual way, and thus it conveys the time at which the process turned passive. When the server receives a message it stores the vector time of the message in the position of the sender. Termination is concluded by the server when it has acquired N concurrent vector timestamps.*

In our algorithm a process only reports passivity to the server when it is passive and has no outstanding messages. We can enforce this requirement since we know that information about outstanding messages will be available at the kernel level. When an active user process is ready to turn passive, it will not be considered passive by the control algorithm until all its messages have been delivered. It is natural for the control algorithm to consider a process with outstanding messages as active; a message residing in the buffers might have to be resent, and only active processes should send messages. A user process must receive all pending messages before it is considered passive by the control algorithm.

Establishing the correctness of the algorithm is trivial. Let us first examine the safety requirement. Assume that termination is detected. Then the server must have a vector timestamp from each process, all of which are concurrent. A process only sends its vector time to the server when it is passive and has no outstanding messages. These two facts in combination imply that all processes are concurrently passive and there are no messages in

transit. Hence, the computation is terminated, and the safety theorem is valid. Next we examine the liveness requirement. Assume that the computation is terminated. Then all processes must be concurrently passive. Each of the processes must have sent a timestamped message to the server when it became passive. Hence, the server will eventually acquire N concurrent vector times and conclude termination. Thus, the liveness theorem is valid and the correctness argument is complete.

The solution presented above stands in sharp contrast to most other solutions presented in the literature. They take the "user's approach" to the system, where there is no knowledge of when a message is delivered. As a result, they go to great lengths to track the messages in the system. In [101], weights are passed around as a means of monitoring the messages in the system, and, in [60], send and receive counts are maintained for all processes for the same purpose. The protocol presented in [17] has some similarities to ours in that acknowledgments are used to monitor the messages in the system. However, vector time is not maintained in the system, which forces their algorithm to send a series of polling waves to determine termination. Due to possible activity "in the back of the wave", at least two waves must be sent even if the system is terminated when the first wave is initiated. Much of the literature also eliminates the problem of messages in transit by considering the simpler problem of distributed termination in a system using synchronous communication [30, 29, 82, 103, 58].

5.6.3 Discussion

As presented above, the server in our algorithm maintains N vector timestamps. Termination is inferred when N concurrent timestamps are acquired. This describes the conceptual operation of the algorithm. In practice, it is sufficient to maintain two vectors to deduce

when N concurrent timestamps have been reported[38]. The server stores the highest timestamp reported for each process in the first vector, and a flag for each process in the second vector. A flag value of true indicates that the highest timestamp reported for the process was reported by the process itself. When all the flags are set to true N concurrent timestamps have been received and termination can be concluded. Thus the practical storage requirement for the algorithm is $O(N)$.

A centralized server is used to determine termination in our algorithm. However, the algorithm can be easily modified to run as a token-based algorithm if so desired. In a distributed solution, the token would contain the collection of vector times. The token would be passed along in lazy fashion. The process holding the token would update its token vector time when passive and without outstanding messages. If the token contains N concurrent vector times, then the process would signal termination, otherwise it would pass the token to the next process. Again, in practice the token only needs to contain two vectors.

Our distributed termination detection algorithm is extremely simple and intuitive. The algorithm utilizes low-level information on delivery of messages, available at the kernel level, which allows it to determine termination solely based on vector time. Termination cannot be detected based on a collection of vector times without this, or other, additional information. The correctness of the algorithm relies on the fact that vector time is updated by ACKs. Since our algorithm uses causal relationships introduced by ACKs, vector time must be updated on ACKs to represent the pertinent causality. Figures 5.15 and 5.16 illustrate how vector time updates on ACKs is indeed necessary. In the figures, acknowledgment messages are indicated by dashed lines and messages to the server are indicated by dotted lines. A gray shadow on the time line for a process indicates that the process is passive. Figure 5.15

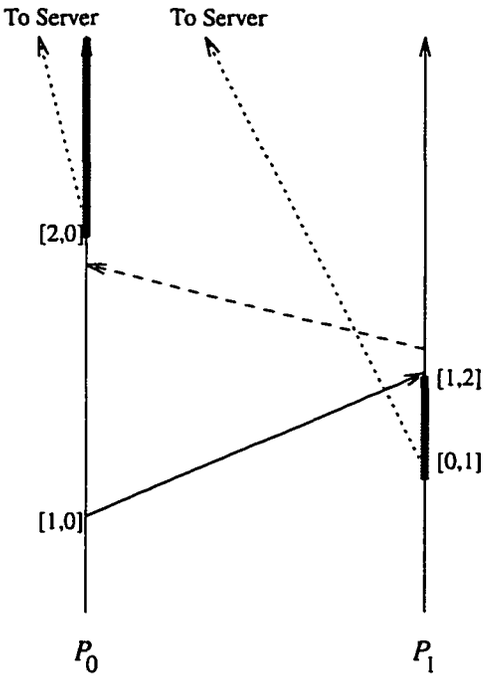

Figure 5.15: ACKs Do Not Update Vector Time

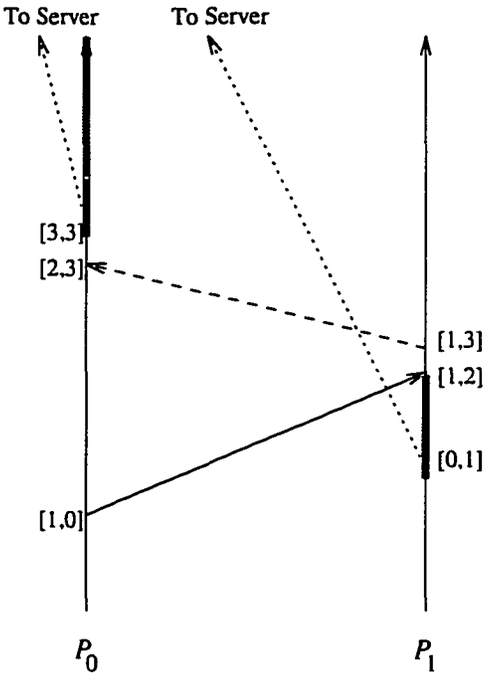

Figure 5.16: ACKs Update Vector Time

shows the computation when ACKs do not update vector time. We can see that the two messages sent to the server reporting passivity are concurrent even though the computation is not terminated. Upon receipt of the timestamped messages the server would erroneously conclude that the computation terminated. In Figure 5.16 the ACKs are allowed to update vector time. We can see that the messages sent to the server now correctly convey the causal relationship between periods of passivity in the two processes. The two messages reporting passivity are not concurrent and thus the server will not deduce termination. If a user-level implementation of the algorithm is desired, then our *delivered_all* construct can be used to propagate information about delivery of messages to the algorithm. However, note that transport layer vector time, with acknowledgment updates, must be used even if

the algorithm is implemented at the user level.

Although simple and intuitive, our distributed termination detection algorithm is interesting not only because it provides a good solution to the problem of distributed termination detection, but also because it illustrates how low-level information on delivery of messages and transport layer vector time with updates on ACKs can be used to design an algorithm that would otherwise not be feasible. Of course, it would be possible to require the control algorithm to send an explicit ACK message for every user message and then use regular vector time. However, an algorithm that requires at least one control message for every user message is not a feasible solution. It is the fact that our solution uses the ACKs already present in the system that makes it a practical solution. Our algorithm illustrates the additional power gained by updating vector time on ACKs and is a good example of how low-level information can be used.

5.7 Chapter Summary

In this chapter we considered the structure of vector time as present at the transport layer as opposed to the application layer. The causal relationship between send and receive events at the transport layer is not necessarily isomorphic to the causal relationship between the events at the application layer. It is therefore crucial to make a distinction between vector time at the two layers. Realizing that the transport layer possesses a more up-to-date view of the system and that many control algorithms will in practice be positioned in kernel space, transport layer vector time is a viable alternative for many algorithms. We began by establishing any formal relationships between application layer and transport layer vector time. We then examined the influence of using transport layer vector time on some algorithms based on vector time from the literature. Algorithms which benefit

from a reduction of the amount of concurrency in the system appears to be well-suited for the use of transport layer vector time. Such algorithms can gain in both efficiency and accuracy through the use of transport layer vector time. However, application layer vector time is more appropriate for algorithms which benefit from concurrency in the system. One advantage of transport layer vector time over application layer vector time is that it provides the possibility to allow acknowledgment messages to update vector time. Utilizing the information flow in the system represented by ACKs can provide us with a more accurate view of the global state of the system. Updating vector time on ACKs is necessary for algorithms that utilize the causal relationships introduced by acknowledgment messages. We derived a distributed termination detection algorithm based on vector time which used the information about message delivery provided by acknowledgment messages. Our algorithm employed causal information propagated through ACKs and thus required vector time to be updated by ACKs. Our distributed termination detection algorithm is a compelling example of how low-level information, available virtually for free but traditionally ignored, can be used to our advantage. As pursued in this dissertation, we believe that the use of information on delivery of messages available at the transport layer offers great possibilities. Examining the causal relationships, as represented by vector time, present at the transport layer as opposed to the application layer is an important step in learning how to better utilize transport layer information.

Chapter 6

Prototype Implementation

In previous chapters we laid the theoretical foundation for the use of transport layer information and discussed some possible applications. Our prototype implementation discussed in this chapter show that our ideas can be put to practical use. This chapter describes the design and implementation of RUPP[1] – a reliable unordered message passing protocol with support for the *delivered* and *delivered_all* primitives. Our implementation of flush channels on top of the RUPP protocol is also considered.

6.1 RUPP Protocol Specification

6.1.1 General Description

The Reliable Unordered Packet Protocol, RUPP, is a connection-oriented protocol designed to provide a reliable unordered message service for distributed applications. The primary target applications are distributed applications running on a local area network. RUPP is designed to provide a simple and efficient service which is easy to implement. In addition

[1] RUPP stands for Reliable Unordered Packet Protocol.

to the basic message service, RUPP supports propagation of information about delivery of messages to the application. This facilitates the implementation of message ordering constraints by the application or by higher-layer protocols. The design of RUPP has been heavily influenced by the Reliable Data Protocol (RDP)[106, 71]. The design was also influenced by SRMP[68], another reliable message passing protocol, as well as by the Transmission Control Protocol[80]. The main features of RUPP are outlined below.

- RUPP provides lightweight connection management. A new connection is implicitly established by the first message sent. This provides efficient connection management for short-lived connections. RUPP connections support bidirectional data transmission with an independent flow of messages going in each direction.

- RUPP presents the application with a reliable message service. Unlike stream-oriented protocols, like TCP, the message boundaries are preserved by RUPP. Reliable message transport is achieved by using sequence numbers and a positive acknowledgment and retransmit policy. RUPP uses a selective retransmit policy where only lost messages are retransmitted. This saves bandwidth compared to a protocol (such as Go-Back-N[100]) that retransmits all messages or segments sent after a lost transmission.

- The message service provided by RUPP is unordered. Messages are not guaranteed to be delivered to the receiving application in the order they were sent. RUPP passes messages to the application in the same order they arrive to the RUPP protocol at the receiving host. Messages arriving on a RUPP connection are immediately available for receipt by the application. As long as there is at least one message available in the receive buffers, the receiving application is never delayed by message loss or message reorder. For a sequenced protocol, on the other hand, the receiving application must

wait for a lost message to be retransmitted even when there is out-of-sequence data present in the receive buffers.

- RUPP provides some simple mechanisms for flow control. It limits the number of packets that can be outstanding on a connection at any given time as well as the maximum skew in sequence numbers allowed between outstanding packets. RUPP also implements a means for notifying the other end of a RUPP connection when there is no receive buffer space left. The other end then refrains from sending additional messages until it is notified that receive buffer space is again available.

- RUPP supports the *delivered* and *delivered_all* primitives. It allows an application to gain information about the safe delivery of messages to the RUPP protocol layer at the receiving host. To the best of our knowledge this feature is unique to the RUPP protocol.

Each of the features outlined above will be described in more detail in subsequent sections.

6.1.2 Relation to other protocols

The RUPP protocol is a transport layer protocol. It was developed with the TCP/IP protocol suite in mind. The position of the RUPP protocol in the TCP/IP protocol suite is illustrated in Figure 6.1. When implemented as a member of the TCP/IP protocol suite, RUPP relies on the IP protocol[78] to provide it with an unreliable datagram service. As part of this service the IP protocol performs protocol demultiplexing and supplies part of the address information needed by RUPP. Although we present RUPP in the context of the TCP/IP protocol suite, it is possible to implement RUPP in a different context as long as the encapsulating protocol presents a sufficient service, similar to the service

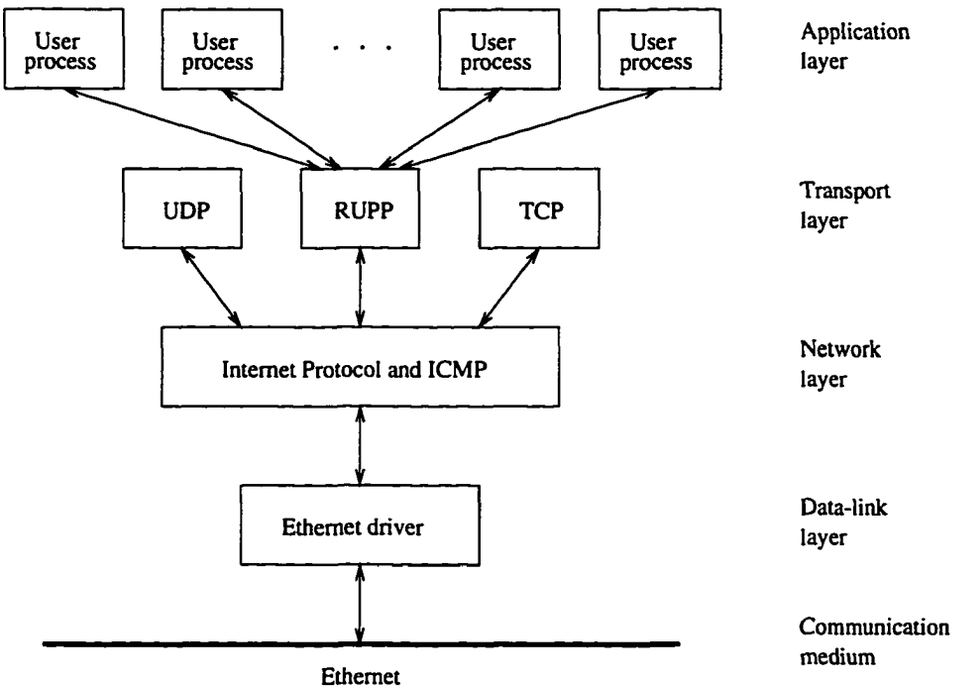

Figure 6.1: Relation to Other Protocols

provided by the IP protocol.

When implemented as part of the TCP/IP protocol suite, RUPP can provide a viable alternative to the traditional protocols (UDP and TCP). In addition to supporting the unique feature of allowing the sender to obtain information about delivery of messages, the RUPP protocol adds reliable delivery to the service provided by UDP without the complexity of TCP. For applications that need reliable service but do not require an ordered byte stream, TCP is unnecessarily complex. The TCP protocol spends considerable effort on providing the data stream abstraction to the application. In addition, TCP does not acknowledge a data byte until all earlier bytes have been received which can lead to unnec-

essary retransmissions[2]. TCP also enforces sequenced delivery, which can slow down the application and consume buffer space when packets are lost or delayed.

The RUPP protocol is most closely related to RDP (Reliable Data Protocol). RDP provides reliable, optionally unordered, message passing. It was proposed by Velten, Hinden, and Sax as an alternative to TCP for packet-based applications such as remote debugging and loading. As described above, TCP has several drawbacks for packet-based applications. These drawbacks served as motivation for the development of RDP. At the moment, RDP is not considered a standard Internet protocol. The latest version of the Internet Official Protocol Standard[76] still lists its state as experimental. The RUPP protocol and RDP differ in their connection management and in how messages are acknowledged. RUPP uses implicit connection setup whereas RDP uses a three-way handshake to set up a connection. Messages that arrive out of sequence are acknowledged using a bitvector in RUPP. RDP acknowledges out-of-sequence messages by including their sequence numbers in its variable-length header.

6.1.3 Header Format

The RUPP protocol adds a header containing control information to each data message. The RUPP header combined with the data message forms a RUPP segment. Additional headers are added to the packet by the lower layers before it is transmitted over the network. The header used by the RUPP protocol has a fixed size of 24 bytes. The format of the RUPP header is shown in Figure 6.2. The header fields are described below.

[2]The fast retransmit and fast recovery algorithms proposed by Jacobson[42] attempt to remedy this problem.

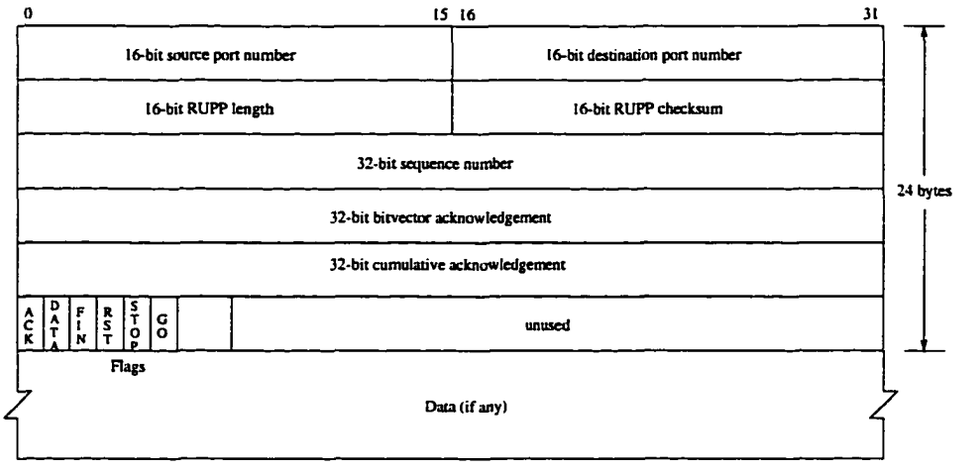

Figure 6.2: RUPP Header Format

Source Port: The 16-bit source port number, in combination with a 32-bit source IP address provided by the IP layer, identifies the communication endpoint on the sending host.

Destination Port: The 16-bit destination port number, in combination with a 32-bit destination IP address provided by the IP layer, identifies the communication endpoint on the destination host. The source and destination port numbers, in combination with the source and destination IP addresses, uniquely identify a connection.

Length: A 16-bit field giving the length of the message in bytes. The length of the header is not included in the length.

Checksum: A 16-bit checksum to allow the receiver to detect corrupted packets. Both the RUPP header and the message data are covered by the checksum.

Flags: This 8-bit field encodes information about the message or the state of the connection. Only 6 of the bits are currently in use. The currently defined flags and their

Mask	Flag	Description
1	ACK	The acknowledgment fields are valid.
2	DATA	The segment contains data.
4	FIN	The other end has closed the connection.
8	RST	Reset, sent in response to a segment arriving on an unavailable communication endpoint.
16	STOP	Receive buffers are are full. Stop sending.
32	GO	Receive space has opened up.

Table 6.1: RUPP Flags

interpretation are displayed in Table 6.1. In our description of the RUPP protocol we will frequently refer to a message with a particular flag set by the name of the flag. For example, a message with the FIN flag set is commonly referred to as a FIN message or simply a FIN.

Sequence Number: If the DATA or FIN flag is set, this field contains the 32-bit sequence number of the message. The sequence number field has no meaning when the message carries no data. (Acknowledgment messages can be sent without data.)

Cumulative Acknowledgment: When the ACK flag is set, this field contains the 32-bit sequence number of the last message received in sequence.

Bitvector Acknowledgment: When the ACK flag is set, this field contains a 32-bit bitvector identifying any messages that have been received out of sequence.

6.1.4 Connection Management

RUPP is a connection-oriented protocol. Connections in RUPP are full duplex. A connection between two processes provides for an independent sequence of messages to be passed

in each direction. Each endpoint of a connection is identified by a host IP address and a port number. A pair of host IP addresses/port numbers forms a connection identifier that uniquely identifies a connection. Several connections can be established between two hosts at any given time.

Connection Setup

Connection setup in RUPP is lightweight. No special messages are passed to establish a connection. The RUPP protocol considers a connection to be established once the two endpoints of the connection are fully specified. As soon as both endpoints are specified, the RUPP protocol changes the state of the connection to ESTABLISHED. Since the communication endpoints can be specified locally, it is possible to establish a RUPP connection with a destination that is unavailable. As soon as the application attempts to transmit data over the connection this will be detected and the error reported to the application.

To establish a RUPP connection the user issues an open request. Open requests can be active or passive. Normally, one side of the connection would perform an active open, and the other side would perform a passive open. An active open request must specify the destination IP address and port number. An active open request changes the state of the connection to ESTABLISHED. An active open can be performed explicitly or implicitly. An explicit open request specifies the destination endpoint without requesting any data to be sent. An implicit open is performed by requesting a message to be sent on a previously unconnected local communication endpoint. A passive open request only specifies the local endpoint of a connection. A passive open request changes the state of the connection to LISTEN. A connection in the LISTEN state changes its state to ESTABLISHED when a packet destined for the local endpoint arrives.

Connection record

The RUPP protocol maintains a connection record for each connection. A few of the more prominent variables maintained in a RUPP connection record are listed below. We do not attempt to give a full list of variables or specify a particular implementation. The list is merely an attempt to illustrate the kind of information that needs to be maintained. Defining some variables will also simplify the further description of the protocol.

Connection Identifier: The source and destination port and IP addresses used to uniquely identify a connection.

state: The current state of the connection. The valid states for a RUPP connection are described in the next subsection.

packets_out: The current number of outstanding packets.

congestion_window: The maximum number of outstanding packets allowed.

max_window: The maximum sequence number difference allowed between two outstanding packets.

sent_seqno: The highest sequence number transmitted.

write_seqno: The highest sequence number sent by the application. If flow control prohibits transmission of a packet, the RUPP protocol will attempt to buffer the message. The write_seqno is always larger than or equal to the sent_seqno.

rcv_seqno: The highest consecutive sequence number received.

rcv_bitvector: A bitvector indicating any messages that have been received out of order. If no messages were ever received out of order, then the bitvector would always

contain all 0's.

rcv_acked_seqno: The highest consecutive sequence number acknowledged by the receiver.

Connection States

During its lifetime, a RUPP connection passes through a sequence of states. The valid states for a RUPP connection are described below.

CLOSED: The CLOSED state is the initial state from which all RUPP connections are created. The CLOSED state exists before the connection identifier is specified. In the CLOSED state, neither of the two communication endpoints that form the connection identifier has been initialized. Other parts of the connection record may be defined.

LISTEN: The LISTEN state is entered when the user performs a passive open. The local endpoint of the communication identifier is defined in the LISTEN state. The connection is waiting for a message to arrive, which implicitly specifies the remote endpoint for the connection.

ESTABLISHED: In the ESTABLISHED state both endpoints of the connection have been defined and user data may flow in both directions. A connection is in the ESTABLISHED state during normal operation.

LOCAL_CLOSING: The LOCAL_CLOSING state is entered when the application issues a close request on an ESTABLISHED connection. No operations can be performed by the application on a connection in the LOCAL_CLOSING state. When the LOCAL_CLOSING state is entered a FIN message is sent to inform the other end about the change in state.

REMOTE_CLOSING: When a message with the FIN flag set is received the RUPP protocol sends an acknowledgment and enters the REMOTE_CLOSING state. Any attempt by the application to send on a connection in the REMOTE_CLOSING state returns an error. Buffered data may still be received by the application. If there is no buffered data an error must be returned. The application is also allowed to receive information about delivery of a message. If the message has not been delivered an error must be returned.

TIME_WAIT: A connection in the LOCAL_CLOSING state enters the TIME_WAIT state when it receives an acknowledgment for its FIN message or it receives a FIN message from the other end. The TIME_WAIT state is entered from the REMOTE_CLOSING state when the application issues a close. A connection in the TIME_WAIT state discards all incoming messages except for incoming FINs. FIN messages are still acknowledged in the TIME_WAIT state to allow the other end to proceed with the closing of the connection.

The state transition diagram for the RUPP protocol is shown in Figure 6.3.

Closing a Connection

Both directions of a RUPP connection are closed simultaneously. The RUPP protocol begins closing down a connection when the application issues a close or when a FIN message is received from the other end. When a close is performed by the application, all data in the send and receive buffers are deallocated. The RUPP protocol does not ensure that all outstanding data is delivered before the connection is closed. If needed, it is up to the application to ensure that all data is delivered before issuing a close request; leaving this responsibility to the application is consistent with the end-to-end argument. The RUPP

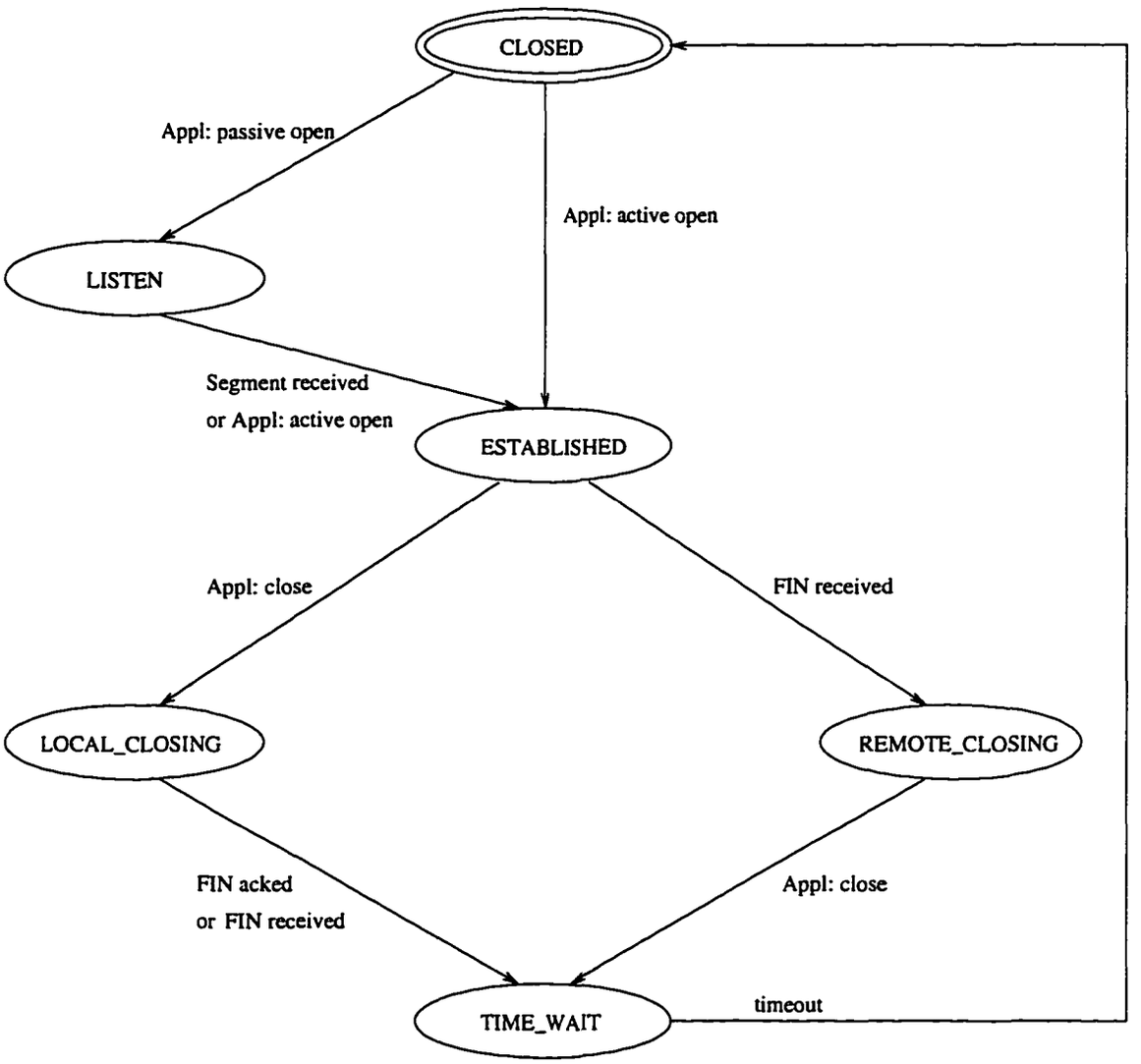

Figure 6.3: RUPP State Transition Diagram

protocol supports propagation of information about delivery of messages to the application which allows the application to carry out this responsibility without appreciable overhead. Many applications can also deduce that all data has been delivered based on a request-response communication pattern.

When a close is issued for a connection in the ESTABLISHED state, the state of the connection is changed to LOCAL_CLOSING and a FIN message is sent to inform the other end. The FIN is delivered reliably to avoid leaving the other end of the connection dangling, possibly waiting for data. RUPP will attempt to retransmit the FIN until it has been acknowledged or until a FIN is received from the other end. The state of the connection is then changed to the TIME_WAIT state. A connection in the TIME_WAIT state discards all incoming segments except for FIN segments. A FIN from the other end is still acknowledged to let the other side continue its close down of the connection. The purpose of the TIME_WAIT state is to prohibit replay errors from occurring. A replay error occurs when a segment from an old connection arrives and causes a new connection to be established or is falsely believed to be part of a current connection. A connection in the TIME_WAIT state filters out any delayed old segments which could otherwise cause replay errors to occur. A connection stays in the TIME_WAIT state for a set amount of time, after which all resources for the connection are deallocated.[3]

A FIN segment is required to carry valid acknowledgment fields. When a connection in the ESTABLISHED state receives a FIN segment, it processes the acknowledgment information contained in the segment. It cancels any scheduled timer events, but leaves its receive and send buffers intact. The state of the connection is changed to REMOTE_CLOSING

[3]To avoid reply errors, the time spent in the TIME_WAIT state should be at least twice the maximum segment lifetime.

and an acknowledgment for the FIN transmitted. The application can still receive buffered data on a connection in the REMOTE_CLOSING state. If no buffered data is available, a receive request results in an error being reported to the application. The application can also query about delivery of messages on a connection in the REMOTE_CLOSING state. This is the reason the FIN segment is required to contain an acknowledgment. If the message segment is still in the send buffers, an error is reported to the user. All attempts to send on a connection in the REMOTE_CLOSING state results in an error being reported to the user. A connection remains in the REMOTE_CLOSING state until the application issues a close request. The connection then enters the TIME_WAIT state.

6.1.5 Data Transfer - Overview

A RUPP connection provides bidirectional data transfer. Data is passed between the RUPP protocol layer on two hosts in the form of RUPP segments. A RUPP segment is formed by prepending a RUPP header to the user data. As RUPP segments are created, they are queued as input for the IP layer. Each segment is kept by the RUPP protocol software until it has been acknowledged. When an incoming packet arrives, it is acknowledged and queued for receipt by the application. If an incoming packet is destined for a communication endpoint which is unavailable, an RST segment is returned to the sender, and the segment is discarded. A detailed description of the measures taken by the RUPP protocol to ensure reliable delivery is given in the next section.

Each message sent by the application is transmitted in a single RUPP segment. The RUPP protocol does not provide fragmentation of user data. Instead, the RUPP protocol imposes a fixed upper limit, MAX_SIZE, on the size of a user message. The MAX_SIZE value is currently set to 1456 bytes. This value was chosen to permit efficient transmission

of the message over an Ethernet. Adding the RUPP header and the IP header to a 1456 byte application message results in a 1500 byte IP datagram. The maximum transmission unit (MTU) on an Ethernet is 1500 bytes[4], thus allowing the IP layer to send the datagram without fragmentation. Any implementation of the RUPP protocol must be able to receive a RUPP segment of MAX_SIZE size. An implementation may choose to enforce a smaller value on the maximum message size it accepts for transmission.

RUPP does not support "keep-alive" segments. Keep-alive segments are used to periodically probe the other end of an idle connection to verify that it is still active. Whether keep-alive segments should be used or not is a controversial issue. Keep-alive segments are not included in the TCP specification since they can cause an otherwise good connection to be terminated due to transient network failure, they waste bandwidth, and they could cost money on connections that are charged by the packet[16]. However, most implementations of TCP support keep-alive segments. Consistent with the end-to-end argument, the RUPP protocol leaves it to the application to poll the other end of a connection if needed.

6.1.6 Reliable Data Transfer

The RUPP protocol provides reliable message transfer to the application. Datagrams transmitted on the communication system underlying the RUPP protocol may be damaged, delayed, lost or duplicated. RUPP uses a combination of mechanisms to ensure reliability on an unreliable communication system. RUPP uses a checksum algorithm to detect damaged segments, sequence numbers are used to detect duplicate segments, and lost segments are resent based on an acknowledgment retransmit mechanism. Each one of the reliability

[4]The Ethernet encapsulation of IP datagrams, defined in RFC 894[81], states that the maximum size of an IP datagram that can be contained in an Ethernet frame is 1500 bytes. Every Internet host connected to a 10 Mbits/sec Ethernet cable is required by the Host Requirement RFC[16] to use RFC 894 encapsulation as its default.

measures used by the RUPP protocol is described in detail below.

Checksum Algorithm

A RUPP segment may be corrupted by the underlying communication system. To ensure the integrity of the transferred data, the RUPP protocol employs a checksum algorithm to detect damaged segments. The RUPP protocol uses the same checksum algorithm as UDP. The RUPP checksum covers both the RUPP header and the data contained in the segment. As for UDP and TCP, RUPP includes a 12-byte pseudo-header in the checksum computation. The pseudo-header contains part of the address information from the IP header. The checksum is calculated as the one's complement sum of 16-bit words. If the number of bytes in the RUPP segment is odd, a pad byte of 0's is added to the end of the segment for the checksum computation. The pad byte is not transmitted. The value of the checksum field of the RUPP header is set to zero during the checksum computation for an outgoing segment. When an incoming segment arrives, the RUPP checksum is calculated. Since the checksum stored by the sender is contained in the checksum calculated by the receiver, the receiver's checksum should contain all ones if the segment is undamaged. Any detected damaged segments are silently discarded.

Sequence Numbers

As is customary for reliable transport protocols, RUPP includes a sequence number in each segment. The sequence number allows detection of duplicate segments and is needed for the acknowledgment retransmit mechanism. Sequence numbers in RUPP are 32 bits long. Thus all arithmetic has to be performed modulo 2^{32}. This will be an implicit assumption in the rest of the presentation.

A separate sequence number stream is used for each direction of the connection. The first segment transmitted in each direction carries the sequence number 1. Using 1 as the initial sequence number for all connections allows us to correctly initialize the connection record before the first segment has arrived. The advantage of this approach is that we do not require the initial segment sent to be the first segment to arrive to the other end. Multiple segments can be transmitted before the initial segment has been acknowledged, improving performance for short-lived connections. A protocol that initializes the connection record based on the initial segment, on the other hand, must wait for the initial segment to be acknowledged before transmitting the next segment.

The drawback of using the same initial sequence number for all connections is that the possibility for replay errors increases. As described above, the requirement to deliver FIN segments reliably in combination with the TIME_WAIT state was designed to avoid replay errors. However, when a host crashes, it is possible that one end of a RUPP connection is left dangling. If the host later recovers and attempts to restart the connection using the same connection identifier, it is possible that the new segments are accepted by the other end as part of the previous connection. If a different initial sequence number is used on each connection, usually assigned based on the processor clock, the probability that the new segment will fall within the sequence number window of the old connection decreases. However, replay errors of this type are very rare and should not be a problem for most applications. In addition, on a local area network, human intervention is often possible to prevent replay errors after a system crash.

A different initial sequence number for each connection, in combination with the use of a three-way handshake protocol for connection setup, is also used to provide a more secure connection. The implicit connection setup and the use of the same initial sequence

number for every connection makes the RUPP protocol vulnerable to IP address spoofing attacks. However, a skillful intruder may be able to predict the initial sequence number and spoof a connection even when a three-way handshake is used for connection setup. A description of the steps involved to spoof a TCP connection, as well as a discussion of other security problems with the TCP/IP protocols, is given by Bellovin[10]. To provide a truly secure connection some form of encryption scheme may have to be used. For our initial development of the RUPP protocol, security is not a major concern. If the RUPP protocol were to be put in widespread use for high security applications, the security issues of the protocol would have to be reexamined. For RUPP applications running on a local area network it may suffice to protect the local network by an IP firewall[37].

Detecting Duplicate Segments

Duplicate segments are detected based on their sequence numbers. To detect duplicates, the RUPP protocol maintains the rcv_seqno and rcv_bitvector, described above, as part of its connection record. Any incoming segment with a sequence number less than or equal to the current value of rcv_seqno is discarded as a duplicate. Since messages may arrive out of order, it is possible that an incoming segment is a duplicate even if its sequence number is greater than the current value of rcv_seqno; in a sequenced protocol the receive buffers can be examined to detect such duplicate segments. Since the RUPP protocol passes messages to the application in the order they arrive, the RUPP protocol cannot rely on the contents of its receive buffers to detect out-of-order duplicates. Instead the RUPP protocol maintains a receive bitvector, rcv_bitvector, which records any messages that are received out of order. Each sequence number is mapped to a specific bit in the receive vector by performing a mod 32 operation on the sequence number. The 32-bit length of the receive vector results

in a maximum allowed value of 32 for the max_window parameter. If a higher sequence number difference between outstanding packets were allowed the receive vector could not be interpreted unambiguously. When a segment arrives with a sequence number larger than rcv_seqno, its corresponding bit in the rcv_bitvector is checked to determine if the segment is an out-of-order duplicate. As the rcv_seqno value is updated, the rcv_bitvector must be updated accordingly.

Figure 6.4 illustrates how RUPP duplicate detection is performed. It shows how the rcv_seqno and rcv_bitvector are updated in response to a series of incoming RUPP segments and the tests performed to detect duplicates. Rather than displaying each bit in the bitvector we display the value of the bitvector in hexadecimal notation. For example, a bitvector with bits 0 and 2 set are displayed as a value of 5. We can see in Figure 6.4 that the second copy of segment 4 is detected as a duplicate since its sequence number is less than or equal to the rcv_seqno. The second copies of segment 3 and segment 6 are detected as duplicates since their corresponding bits in the rcv_bitvector are already set. Also note that the rcv_bitvector is updated in accordance with the rcv_seqno. After segment 5 arrives there are no out-of-order segments received and the rcv_bitvector is 0.

Acknowledgment and Retransmission

RUPP segments may be lost or damaged by the underlying communication system. To ensure reliable delivery of messages to the application, lost segments must be retransmitted. RUPP provides a simple acknowledgment retransmit mechanism. RUPP keeps a timer for its oldest outstanding segment. If the timer expires before an acknowledgment is received the segment is retransmitted. RUPP uses a fixed initial timeout value, RETR_TIMEOUT, for its retransmit timer. RUPP does not attempt to measure the round-trip time (RTT)

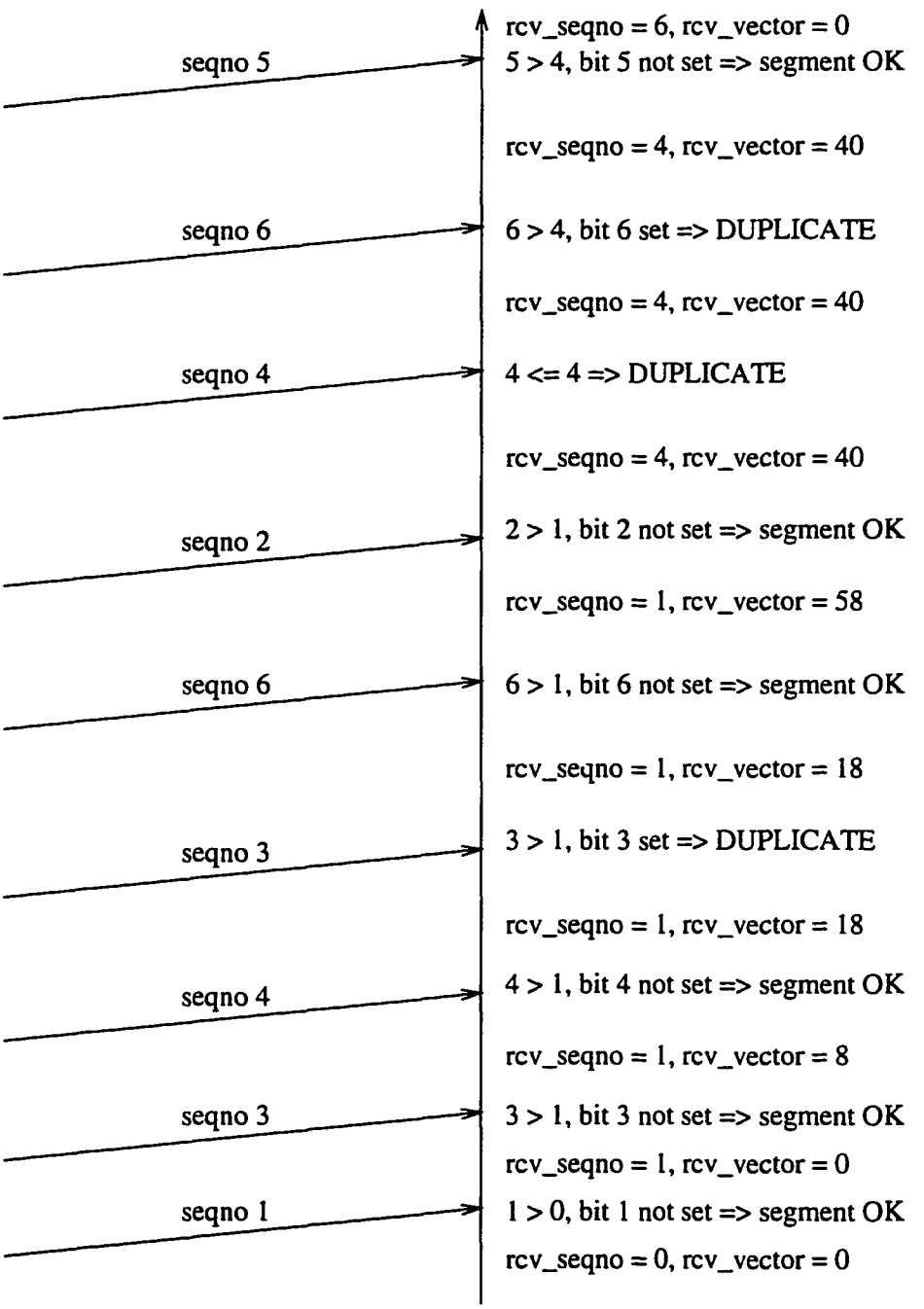

Figure 6.4: RUPP Duplicate Detection

experienced on a connection, as is done in TCP. Over a local area network we expect the variability in round-trip time to be low. To provide resilience in the face of transient network errors or temporary congestion RUPP uses an exponential timer back-off[41]. The timeout interval is doubled between successive retransmit attempts. The RUPP protocol will make a fixed number of attempts, MAX_ATTEMPTS, at transmitting a message before giving up and reporting an error to the application.

When an incoming segment arrives, it must be acknowledged. By default RUPP will delay an acknowledgment in anticipation of being able to piggyback the ACK on user data going in the other direction. An acknowledgment is delayed for a maximum time interval, ACK_TIME. By default, no more than one unacknowledged segment is allowed. If a second segment arrives within the ACK_TIME interval, an acknowledgment is immediately sent. RUPP supports an option for acknowledging incoming segments without delay. RUPP also supports an option for changing the number of allowed unacknowledged segments. A RUPP acknowledgment consists of two parts, a cumulative acknowledgment and a bitvector acknowledging segments received out of sequence. A RUPP acknowledgment is constructed by including the rcv_seqno as the cumulative acknowledgment and the rcv_bitvector as the bitvector acknowledgment in the RUPP header.

In addition to the basic acknowledgment mechanism described above RUPP supports a fast retransmit algorithm. The fast retransmit algorithm is an adaptation of the fast retransmit algorithm proposed for TCP by Jacobson[42]. TCP is required to send an immediate ACK when an out-of-order segment arrives. Since TCP uses only cumulative acknowledgments, out-of-order segments will cause duplicate ACKs to be transmitted. The purpose of these ACKs is to let the other side know that an out-of-order segment has been received and what sequence number is expected next. An out-of-order segment can

arrive due to a reordering of segments or as the result of a lost segment. If a reordering of segments occurred, we expect no more than one or two duplicate ACKs to arrive before the reordered segment is processed. If additional duplicate ACKs are received in a row, this provides strong evidence that a segment has been lost. When three or more duplicate ACKs are received in a row, the fast retransmit rule of TCP responds by immediately retransmitting the missing segment without waiting for the retransmit timer to expire. The fast retransmit rule used by TCP is easily adopted for use by the RUPP protocol. Since RUPP acknowledges out-of-order segments, no duplicate ACKs can be expected. Instead, the RUPP protocol immediately retransmits a missing segment when three or more out-of-order segments have been acknowledged. Note that this information could be carried in a single RUPP acknowledgment. The number of acknowledged out-of-order segments can be derived as sent_seqno − rcv_acked_seqno − packets_out.

6.1.7 Flow Control

To avoid overwhelming the receiver, a reliable transport protocol must provide mechanisms for flow control. RUPP relies on a simple window-based scheme for flow control. The number of outstanding packets on a connection is controlled by two variables, the max_window and the congestion_window. The max_window specifies the maximum skew allowed between the sequence numbers of any two outstanding packets. As discussed above, the maximum possible value for max_window is 32. This is the recommended value. With the fast retransmit rule used by RUPP is it unlikely that the transmission of a segment over an Ethernet LAN is ever suspended due to the max_window. The congestion_window, discussed next, would normally prevent transmission of additional segments before the max_window is exhausted. The congestion_window specifies the maximum number of outstanding packets allowed on

a given connection. The congestion_window value should be chosen to allow maximum throughput without causing receive buffer overflow; since the size of the transmitted RUPP segments depend on the size of the data messages sent by the application, there is no clear choice. We recommend a value of 15 as a reasonable trade-off between throughput requirements and buffer space requirements. Based on a bandwidth-delay product[5] of 3,750 bytes for an Ethernet LAN[98], a congestion_window of 15 will allow segments with an average size of 250 bytes or more to be sent without delay. When bulk data transfer is performed we would expect the transmitted segments to be at least this size. Fifteen RUPP segments with a maximum size of 1456 bytes occupy 21840 bytes of buffer space. Before transmitting a packet, the RUPP protocol checks that the resulting number of outstanding packets does not exceed the congestion_window value and that the maximum skew in sequence numbers does not exceed the max_window value. If transmission of a segment violates either of the flow control requirements, then transmission of the segment is suspended. Suspended segments are transmitted later when an incoming ACK opens up the windows.

The flow control mechanism described above does not guarantee that the receive buffers do not overflow. Recall that a RUPP segment is acknowledged when it has been correctly received by the RUPP protocol layer at the other end of the connection. Incoming segments are queued to be delivered to the application. Thus, even with a congestion_window value of 1 it is possible to exhaust receive buffer space if the application is not issuing receives. For a reliable transport protocol it is not sufficient to simply discard incoming segments if there is no buffer space available. This would cause the sending side to repeatedly retransmit a segment if no buffer space became available. This not only wastes bandwidth,

[5]The capacity of a pipe is given by its bandwidth-delay product. It is calculated as bandwidth(bits/sec)*round-trip time(sec).

but could cause the sender to falsely detect a network failure and report an error to the application. Handling receive buffer overflow properly is, thus, more a matter of reliability than performance. It is not acceptable for a reliable transport protocol to lose segments as a result of the receiving application lagging behind.

The RUPP protocol uses a simple stop-and-wait algorithm to handle receive buffer overflow. If an incoming segment must be discarded due to buffer overflow, the RUPP protocol informs the sender by transmitting a RUPP segment with the STOP flag set. When sufficient buffer space is reclaimed to allow receipt of at least one maximum sized segment, the RUPP protocol sends a segment with the GO flag set. When a segment with the STOP flag set is received, segment transmission is suspended until a message with the GO flag set arrives. To avoid deadlock in case the GO segment gets lost, RUPP enters a probing phase when a segment with the STOP flag set is received. During probing the oldest outstanding segment is transmitted at regular intervals. These retransmit attempts do not count towards the number of retransmit attempts performed before giving up on a segment. If a new acknowledgment is received during probing, it is considered an implicit GO indication. Segments with the STOP flag set are required to carry valid acknowledgment fields.

6.1.8 Information About Delivery of Messages

RUPP supports the propagation of information about delivery of messages to the application. The application may inquire about a specific message or inquire whether all messages have been delivered. When the application requests information about a specific message it must provide the *sequence number offset* of the message. The sequence number offset for the nth message sent on a RUPP connection is $n - 1$. The RUPP protocol then maps this offset to a segment sequence number. If the sequence number is less than or equal to the

current value of rcv_ack_seqno, the message has been delivered. If the sequence number is larger than rcv_ack_seqno, the RUPP protocol examines its send buffers for the message. If the message is still in the buffers, the application is suspended until the message has been delivered or an error occurs. An application that requests information about delivery of all messages sent is suspended until the RUPP send buffers are empty or an error occurs.

The requirement to provide information about delivery of messages to the application is unique to the RUPP protocol. The primary motivation for developing RUPP was to show practical evidence of how information about delivery of messages can be utilized at the application level. The requirement to provide this information to the application is therefore part of the specification of RUPP. Adding this feature to other reliable message passing protocols is straightforward. Providing information about the delivery of user data is most intuitive for message-based protocols, but this feature could also be added to stream-oriented protocols such as TCP. For a stream-oriented protocol, the user would request information about the delivery of all data up to a specified byte or about the delivery of all sent bytes.

6.1.9 User Interface

The specification of RUPP given above is independent of the user interface. The user interface provided for RUPP is implementation dependent. This section merely indicates the kind of functionality that should be provided.

Open Request: The open request is used to establish a connection. Open requests may be active or passive. An active open request must contain the destination port number and IP address.

Send Request: The send request is used to send an application message. The data buffer and the length of the message must be specified. A send request which specifies the destination endpoint provides implicit connection establishment.

Receive Request: The receive request is used to receive an application message. The data buffer and the length of the message must be specified.

Delivery Request: The delivery request is used to query the RUPP protocol about delivery of messages. A query should be allowed on a specific message or on all messages. When the query refers to a specific message, the sequence number offset of the message must be provided.

Close Request: The close request is used to close a connection.

6.2 RUPP Prototype Implementation

Our prototype implementation of RUPP was carried out within the realm of the Linux operating system[108]. Linux is a complete UNIX clone for personal computers. Linux currently runs on Intel 386, 486, and Pentium machines. The full source code for the Linux operating system is freely available, making it an excellent source for experimental system development.

As in most UNIX systems, the TCP/IP networking code in Linux is part of the kernel. The application programming interface to TCP/IP is provided through Berkeley sockets. Our implementation of RUPP was incorporated into version 1.2.0 of the Linux kernel. Any further references to the Linux kernel implicitly refers to version 1.2.0 of the kernel. The Linux kernel is rapidly changing, thus the description of the networking code given below may not be completely accurate for other versions of the kernel.

6.2.1 The Networking Code in Linux

Before a Linux application can send or receive a message it must create a socket using the *socket* system call. The *domain*, the *type*, and, optionally, the *protocol* of the socket must be supplied as parameters to the call. The *domain* specifies the communication domain for the socket; this defines the address family used for operations on the socket. The domain for TCP/IP sockets is AF_INET. The *type* of the socket specifies the semantics of communication. For example, a socket of type SOCK_STREAM provides a reliable, sequenced, connection-oriented byte stream. The *protocol* specifies the particular protocol to be used. Generally, only a single protocol is implemented for each type of socket and the protocol does not need to be specified. TCP is the default protocol for sockets of type SOCK_STREAM, and UDP is the default protocol for sockets of type SOCK_DGRAM.

Each supported address family and protocol is specified in the Linux networking code as a collection of functions. The `proto` structure which specifies an protocol in the AF_INET domain is shown in Figure 6.5. The functions specified in the `proto` structure defines the interface to a protocol. In addition, a protocol may implement numerous functions for internal use. The `proto_ops` structure which is used to specify an address family have a similar format.

The easiest way to understand how the networking code in Linux is structured is to look at what happens when an application sends a message. We will assume that the application has created a socket of type SOCK_DGRAM in the AF_INET domain. We will examine what happens when the application sends a message using the *sendto* system call. The system call is vectored through a stub in *libc* which sets up the calling stack frame and traps into the kernel. All system calls enter the kernel through the same entry point, the `_system_call` function. Control is transferred to the actual code for the system call based

```
struct proto {
  struct sk_buff *    (*wmalloc)(struct sock *sk, unsigned long size,
                             int force, int priority);
  struct sk_buff *    (*rmalloc)(struct sock *sk, unsigned long size,
                             int force, int priority);
  void                (*wfree)(struct sock *sk, struct sk_buff *skb,
                             unsigned long size);
  void                (*rfree)(struct sock *sk, struct sk_buff *skb,
                             unsigned long size);
  unsigned long       (*rspace)(struct sock *sk);
  unsigned long       (*wspace)(struct sock *sk);
  void                (*close)(struct sock *sk, int timeout);
  int                 (*read)(struct sock *sk, unsigned char *to,
                             int len, int nonblock, unsigned flags);
  int                 (*write)(struct sock *sk, unsigned char *to,
                             int len, int nonblock, unsigned flags);
  int                 (*sendto)(struct sock *sk, unsigned char *from,
                             int len, int noblock, unsigned flags,
                             struct sockaddr_in *usin, int addr_len);
  int                 (*recvfrom)(struct sock *sk, unsigned char *from,
                             int len, int noblock, unsigned flags,
                             struct sockaddr_in *usin, int *addr_len);
  int                 (*build_header)(struct sk_buff *skb, unsigned long saddr,
                             unsigned long daddr, struct device **dev,
                             int type, struct options *opt, int len,
                             int tos, int ttl);
  int                 (*connect)(struct sock *sk,
                             struct sockaddr_in *usin, int addr_len);
  struct sock *       (*accept) (struct sock *sk, int flags);
  void                (*queue_xmit)(struct sock *sk, struct device *dev,
                             struct sk_buff *skb, int free);
  void                (*retransmit)(struct sock *sk, int all);
  void                (*write_wakeup)(struct sock *sk);
  void                (*read_wakeup)(struct sock *sk);
  int                 (*rcv)(struct sk_buff *buff, struct device *dev,
                             struct options *opt, unsigned long daddr,
                             unsigned short len, unsigned long saddr,
                             int redo, struct inet_protocol *protocol);
  int                 (*select)(struct sock *sk, int which,
                             select_table *wait);
  int                 (*ioctl)(struct sock *sk, int cmd,
                             unsigned long arg);
  int                 (*init)(struct sock *sk);
  void                (*shutdown)(struct sock *sk, int how);
  int                 (*setsockopt)(struct sock *sk, int level, int optname,
                             char *optval, int optlen);
  int                 (*getsockopt)(struct sock *sk, int level, int optname,
                             char *optval, int *option);
  unsigned short      max_header;
  unsigned long       retransmits;
  struct sock *       sock_array[SOCK_ARRAY_SIZE];
  char                name[80];
  int                 inuse, highestinuse;
}
```

Figure 6.5: The proto Structure

on an offset in the _sys_call_table. The offset is put in the calling stack frame by the *libc* library routine. Almost all socket system calls use the same offset and thus are vectored to the same function, sys_socketcall. Based on additional information put in by the *libc* stub, sock_sendto is invoked. In sock_sendto, the socket structure associated with the file descriptor provided by the application is retrieved. The socket structure contains information on the address family and protocol associated with the socket. After some error checking, control is passed to the address family by invoking its specified sendto function; in our example this is inet_sendto. Inet_sendto performs additional error checking and then invokes the sendto function specified for the protocol, in our case udp_sendto. The UDP protocol performs the bulk of the work involved in sending the message. In short, it allocates a suitable datagram, fills in the header information, copies the user data, performs checksum computation, and invokes the IP protocol to send the datagram. The IP protocol will in turn invoke the device driver to actually transmit the datagram.

6.2.2 Adding RUPP

As mentioned above, the protocols in the TCP/IP suite are specified as a collection of functions. The first step in implementing RUPP was then to define an appropriate proto structure for the protocol. The proto structure specifying the RUPP interface is shown in Figure 6.6[6]. Although many minor adjustments to existing header files and code were needed to integrate the RUPP protocol, the bulk of the RUPP implementation is contained within functions internal to the protocol or the functions specified in the RUPP proto structure.

RUPP is implemented according to the protocol specification given in the previous

[6]Since our protocol was initially without a name and its specification similar to RDP, we used the name RDP during development of the code

```
struct proto rdp_prot = {
        sock_wmalloc,
        sock_rmalloc,
        sock_wfree,
        sock_rfree,
        sock_rspace,
        sock_wspace,
        rdp_close,
        rdp_read,
        rdp_write,
        rdp_sendto,
        rdp_recvfrom,
        ip_build_header,
        rdp_connect,
        NULL,
        ip_queue_xmit,
        NULL,
        NULL,
        NULL,
        rdp_rcv,
        datagram_select,
        rdp_ioctl,
        rdp_init,
        NULL,
        rdp_setsockopt,
        rdp_getsockopt,
        128,
        0,
        NULL,,
        "RDP",
        0, 0
}
```

Figure 6.6: The RUPP proto Structure

section. The state transition diagram of Figure 6.3 is implemented in software with the exception that the LISTEN state is not used. The operation of the protocol in the CLOSED and LISTEN states is similar enough that a distinction, in software, between the two states would only add complexity to the code. The Linux networking code maintains a `sock` structure for each socket in the inet domain. The RUPP connection record constitutes part of the information stored in the `sock` structure. Our implementation differs from the specification in its treatment of segments that arrive for an unknown port. Our implementation does not use RST segments. Instead, we send an ICMP port unreachable message[79] to inform the other end when a segment for an unknown communication endpoint is received. When an ICMP error message is received by the RUPP software the error is reported to the application.

In addition to the options described in the RUPP specification, we also implemented an option for changing the initial retransmit timeout used on a connection. This additional option was implemented to allow easy experimentation with the protocol. Information about RUPP was incorporated into the `net` directory of the `/proc` filesystem[46]. The `/proc` filesystem in Linux is a virtual filesystem, providing an interface to kernel data structures. It provides easy access to kernel statistics and process information. Applications can obtain information from the `/proc` filesystem instead of having to read `/dev/kmem`. The RUPP protocol maintains statistics on the number of segments sent, number of retransmits, number of segments received, and so on. This information is available through the `/proc` filesystem interface.

Control is transfered to the RUPP protocol software by three different mechanisms: as the result of a system call executed by the application, when a RUPP timer expires, and from the IP layer as a result of a hardware interrupt caused by an incoming packet.

Much of the complexity involved in implementing the RUPP protocol stems from the need to provide adequate concurrency control. System call invocations have the lowest priority. The RUPP software can never be interrupted during processing as a result of an application performing a system call. RUPP maintains three timers for each socket, a retransmit timer, an ack timer, and a waitfor timer. A RUPP timer may expire and interrupt the RUPP protocol software during processing. Functions invoked as a result of a system call can protect themselves from a timer operating on the same socket by setting the `inuse` flag, which is part of the `sock` structure. Processing of an incoming packet is protected from timer operations through the `in_bh` flag which is set by lower layers. Each timer function begins by turning off interrupts and checking if the socket is in use or if processing of an incoming packet is in progress by checking the `inuse` flag for the socket and the `in_bh` flag. If either flag is set, then the timer reschedules itself, turns interrupts on, and exits. If neither flag is set, the timer code sets the `inuse` flag to protect itself from another timer operating on the same socket, turns interrupts back on, and proceeds with processing. Processing of an incoming packet has the highest priority. Unless interrupts are turned off, an incoming packet will interrupt processing performed as a result of a system call or a timer event. Hence, access to volatile variables and other critical operations must be protected in the code by turning interrupts off. For example, suspension of a process must be protected from interrupts to make sure that the "wakeup call" for the process is not missed.

6.2.3 User Interface

As mentioned above, the networking user interface in Linux is provided through sockets. A socket for the RUPP protocol should be created as type SOCK_RDM, a socket for reliably delivered messages. No protocol was previously supported for a socket of type SOCK_RDM

so RUPP is the default protocol for a socket of this type. No protocol number needs to be specified when the socket is created.

All the standard socket system calls supported on a UDP socket are supported by RUPP. An application using UDP which only communicates with one process on each UDP port should be able to move to reliable communication by simply altering the type of its sockets to SOCK_RDM. Since RUPP uses implicit connection setup the *listen* and *accept* system calls, used by TCP, are not supported.

An active open for a RUPP connection is performed explicitly using the *connect* system call or implicitly by sending a message using the *sendto* system call. A passive open is performed by binding the socket to a local port number through the *bind* system call. Messages are transmitted using the *send* or *sendto* system calls and received using the *recv* or *recevfrom* system calls. When *sendto* is used to transmit a message on a connected socket, the destination address must match the address given when the connection was opened. The *read* and *write* system calls are also supported. A RUPP connection is closed using the *close* system call. RUPP options are manipulated through the *setsockopt* and *getsockopt* system calls.

6.2.4 Delivered and Delivered_all

Two new system calls, *delivered* and *delivered_all*, were added to the Berkely socket interface in Linux to retrieve information about delivery of messages from the RUPP protocol. The *delivered* and *delivered_all* system calls are only valid for RUPP sockets. The calls are declared in a header file, `del.h`, as:

int delivered(int s, message_id mid)

int delivered_all(int s)

where *s* should be a RUPP socket and *mid* is the message sequence number offset. The type `message_id` is defined in the header file as unsigned long. As described earlier, the sequence number offset for the *n*th message sent on a connection is $n - 1$.

To implement the system calls two library stubs were created for the calls. As for most socket system calls, we have the kernel entry routine vector the *delivered* and *delivered_all* system calls to the `sys_socketcall` function. Besides the `_sys_call_table` offset and the user parameters, the library stub provides an identifier to indicate what socket call was invoked. Two new identifiers, SYS_DEL and SYS_DELALL, were defined. Based on these identifiers `sys_socketcall` invokes an appropriate function. The type of the socket is checked and some general error checking performed before the RUPP protocol is invoked to handle the calls.

The RUPP software acts in accordance with the RUPP specification. For a *delivered* call the sequence number offset is mapped to a segment sequence number by adding the offset to the initial send sequence number. The resulting segment sequence number is compared against rcv_ack_seqno. If the sequence number is larger than rcv_ack_seqno the RUPP software examines its send buffers for the segment. If the segment is found the calling application is suspended until the segment has been delivered or an error occurs. For a *delivered_all* call the send buffers are examined. If there are outstanding segments the calling application is suspended until all segments have been delivered or an error occurs.

When an error condition occurs on a RUPP socket, it is up to the application to correctly interpret the result of subsequent calls to *delivered* and *delivered_all*. The RUPP software makes a fixed number of attempts at delivering a segment. After MAX_ATTEMPTS retransmit attempts, the RUPP software gives up. The segment is cleared from the buffers and an ETIMEDOUT error is recorded for the socket. The error is reported to the ap-

plication the next time a system call is invoked on the socket. The error is then cleared. At this point the segment that caused the ETIMEDOUT error is indistinguishable from a segment that was successfully delivered. The RUPP protocol does not close a socket when an error occurs. Instead it is up to the application to decide how to proceed after an error was reported on a socket. If the application decides to carry on it must account for the fact that the semantics of later operations on the socket may not be preserved. Allowing the application to decide how to proceed after an error occurs is consistent with the end-to-end argument.

6.3 Flush Channel Implementation

Flush channel communication was introduced in Chapter 4. A straightforward user-level implementation of flush channels was presented. Having implemented the RUPP protocol and the *delivered* and *delivered_all* system calls, it was an easy task to transform our proposed implementation into a flush channel library running on top of RUPP. In the library we defined a new socket type, SOCK_FLUSH, for flush channel semantics. SOCK_FLUSH is simply an alias for SOCK_RDM since the flush channel is built on top of a RUPP socket. The SOCK_FLUSH type was defined in support of good coding style. Two library functions were declared for transmitting a message over a flush channel socket, `flush_send` and `flush_sendto`. As described in Chapter 4, flush channel messages have a type in addition to the data. Thus the `flush_send` and `flush_sendto` routines require a *type* parameter in addition to the regular parameters required by the *send* and *sendto* system calls. The two routines are declared as:

*int flush_send(int s, const void *msg, int len, unsigned int flags, char type)*

*int flush_sendto(int s, const void *msg, int len, unsigned int flags,*

*const struct sockaddr *to, int tolen, char type)*

The code for the `flush_sendto` routine is shown in Figure 6.7. The mapping from the proposed implementation in Chapter 4 to the code in Figure 6.7 is obvious. The `D_FLAG`, `mid`, and `count` variables are declared as static external variables. They are used by both the `flush_send` and `flush_sendto` routine and they must maintain their values between successive invocations of the routines. The `count` variable records the number of messages sent on the flush channel. It gives the sequence number offset needed for the *delivered* call. The `flush_send` routine looks very similar with the *sendto* system calls replaced by *send* system calls.

Since our flush channel implementation puts the sole responsibility of providing the correct semantics on the sender, no flush_receive routine is needed. Applications communicating over a flush channel use all the standard system calls on the socket except for when transmitting a message. All messages transmitted on a SOCK_FLUSH socket must be transmitted using the `flush_send` or `flush_sendto` routine. It is the responsibility of the application to conform to this rule.

6.4 Chapter Summary

This chapter presented our prototype implementation. To illustrate how transport layer information can be used in practice, we designed and implemented the RUPP transport layer protocol. The RUPP specification requires protocol support for propagation of information about delivery of messages to the application. RUPP does not only provide message delivery information, but is a complete transport layer protocol. RUPP provides the application with

```
int flush_sendto(int s, const void *msg, int len, unsigned int flags,
                 const struct sockaddr *to, int tolen, char type)
{
 int res;

 switch( type ) {

      case ORD:
           if(D_FLAG) {
                if ((res = delivered(s, mid)) < 0)
                     return -1;
                D_FLAG = 0;
           }
           res = sendto(s, msg, len, flags, to, tolen);
           count++;
           break;

      case TWF:
           if ((res = delivered_all(s)) < 0)
                return -1;
           res = sendto(s, msg, len, flags, to, tolen);
           D_FLAG = 1;
           mid = count;
           count++;
           break;

      case FF:
           if ((res = delivered_all(s)) < 0)
                return -1;
           res = sendto(s, msg, len, flags, to, tolen);
           D_FLAG = 0;
           count++;
           break;

      case BF:
           if(D_FLAG)
                if ((res = delivered(s, mid)) < 0)
                     return -1;
           res = sendto(s, msg, len, flags, to, tolen);
           D_FLAG = 1;
           mid = count;
           count++;
           break;

      default:
           errno = -EINVAL;
           return -1;
      }
 return res;
}
```

Figure 6.7: The flush_sendto Routine.

a reliable unordered message passing service. Some of the main features of RUPP include: lightweight connection management; reliable delivery of messages based on a checksum algorithm, sequence numbers, a receive bitvector to record segments received out of order, and an acknowledgment retransmit algorithm which includes a fast retransmit rule; window-based flow control; and mechanisms for buffer management.

The RUPP protocol was implemented within the networking code of the Linux kernel (version 1.2.0). The user interface to the RUPP protocol is provided through the standard socket system calls available in Linux. To retrieve information about delivery of messages from a RUPP socket, two new system calls, *delivered* and *delivered_all*, were implemented. The *delivered* and *delivered_all* system calls are only available on RUPP sockets.

Once the RUPP protocol was implemented it was an easy task to implement a flush channel library on top of it. Based on the implementation proposed in Chapter 4 two library routines, `flush_send` and `flush_sendto`, were constructed. All messages transmitted on a flush channel socket must be transmitted through one of these two routines. The regular socket library routines are used for all other operations on the socket.

The implementation described in this chapter show that the improved use of transport layer information is not only a theoretically appealing idea but achievable in practice. Although the requirement to support propagation of information about delivery of messages to the application is unique to the RUPP protocol, it is straightforward to add similar support to other reliable transport protocols.

Chapter 7

Experimental Results

Our prototype implementation of the RUPP protocol and the *delivered* and *delivered_all* system calls were presented in the previous chapter. Our implementation of a flush channel library routine was also discussed. In this chapter we present experimental results obtained from our prototype implementation. The first set of experiments evaluates the performance of RUPP compared to TCP and UDP. The second set of experiments evaluates the performance of our *delivered* and *delivered_all* primitives, and the final set of experiments evaluates the performance of our flush channel implementation. We begin the chapter with a brief description of our experimental setup. Any results presented are naturally bound to the specific Linux implementations of TCP and RUPP that were used. However, it is safe to assume that any performance gains displayed for our implementation are due to inherent advantages of a reliable unordered transport protocol. Although some effort was spent on code optimization, the main focus for our prototype implementation was on functionality and correctness. The design and implementation of TCP is naturally much more mature and more highly optimized.

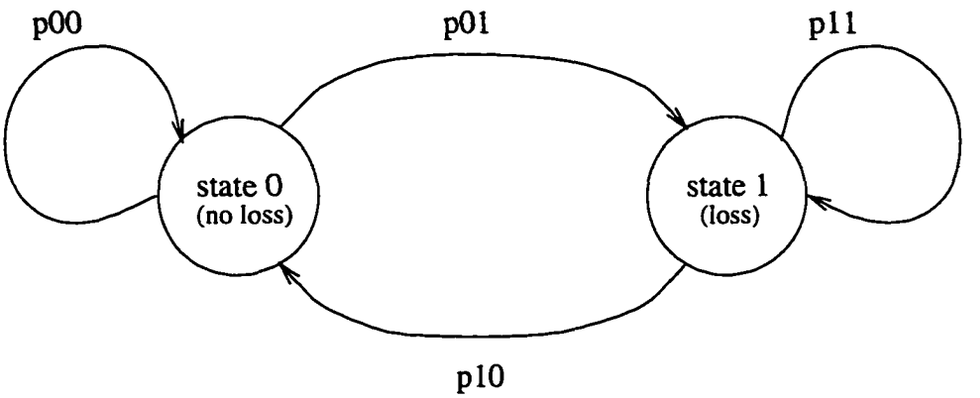

Figure 7.1: Markov Model for Packet Loss

7.1 Experimental Setup

Our experiments were conducted over a 10Mbits/sec local area Ethernet. All the machines used in our experiments were 90MHz Pentiums running the Linux 2.0 kernel. At the time our experiments were conducted, there was minimal competing traffic on the network. The only competing traffic was due to regular system activities such as clock synchronization.

During normal operation, no packets were lost or corrupted during transmission from one host to another. To evaluate performance over an unreliable network, we implemented a socket option to probabilistically discard incoming packets. We refer to a socket with this option enabled as a "lossy" socket. The option was implemented for both TCP and RUPP sockets. A Lehmer random number generator[70] was compiled into the kernel and the value passed in when setting the socket to lossy was used to seed the generator. Packets arriving to a lossy socket were dropped based on a two-state Markov chain, illustrated in Figure 7.1. By default, a transition was performed in the Markov chain each time a packet arrived on the socket. If the Markov chain was in the loss state (state 1) after the transition,

then the packet was dropped. (This is Gilbert's Markov model for channels with memory with an error probability of 1 in the "bad" state and an error probability of 0 in the "good" state[47].) For our experiments with an unreliable network, we dropped packets both at random and in a bursty fashion. Random packet loss was achieved by selecting the state transition probabilities in the Markov chain such that $p00 = p10$ and $p01 = p11$. Thus, the state transition probability of entering the loss state was the same from both states of the Markov chain. As a result, an independent Bernoulli trial was performed for each incoming packet to decide if it should be dropped or not. Bursty packet loss was initially created by modifying the state transition probabilities to increase the expected number of packets lost when the loss state was entered. However, this did not work well. The problem was that both TCP and RUPP respond to packet loss by increasing the retransmit timeout, thus sending fewer packets, but the only way to leave the loss state was to transmit packets. The assumption made by TCP and RUPP that bursty errors are tied to time is reasonable. It was ill matched by the model which created bursty error behavior through the number of packets lost independent of time. To remedy this problem we allowed a holding time to be associated with the loss state. Only when the first packet arrived after the holding time had expired was a state transition performed. This allowed several packets to be lost each time the loss state was entered during normal transmission. The process of selecting the holding time will be discussed further below when we describe our experiments for an unreliable network.

For all experiments, average timings were obtained and the 95% confidence intervals for the average timings were calculated. Where possible, the calculated confidence intervals are displayed in our graphs. Although the 95% confidence intervals produced for the experiments were generally very small, we saw evidence of a higher variability in the system. This

was especially true for the TCP protocol. For our bulk data transfer experiment, described below, repeated executions of the experiment produced clearly non-overlapping confidence intervals for TCP. This suggests that the time required for bulk data transfer over successive TCP connections may not be completely independent. We see no obvious explanation for a dependence between successive TCP connections, and merely account it to the fact that we are dealing with a real system. Although some of the confidence intervals produced for TCP appeared to be misleadingly small, the general trend illustrated by our experiments always remained consistent. The performance measured for the RUPP and UDP protocols was much more consistent than the TCP performance.

7.2 RUPP Experiments

Our first set of experiments was designed to evaluate the performance of our RUPP protocol in comparison with TCP and UDP. All tests for TCP were run with the TCP_NODELAY option enabled. Normally, TCP will delay the transmission of small packets in an attempt to combine several small packets into a single packet before transmission. Setting the TCP_NODELAY option caused TCP to immediately transmit each packet, providing the most useful comparison for our protocol.

7.2.1 Bulk Data Transfer (Experiment 1)

In our first experiment, we examined the performance of the protocols for bulk data transfer. We measured the time it took to transfer a batch of 512-byte messages back and forth between two hosts for varying batch sizes. The batch of messages was passed in both directions since all timings had to be performed on the same host to avoid timing errors due to clock drift between the two machines. The pseudo code for the host controlling the

```
get starting time;
open connection to remote host;
for(i=0; i<batch_size; i++)
     send 512-byte message;
for(i=0; i<batch_size; i++)
     receive 512-byte message;
close connection;
get ending time;
print out time used;
```

Figure 7.2: Outline of Master Process

experiment is shown in Figure 7.2. As seen in Figure 7.2, the time needed to establish the connection was included in the timing. Hence, the experiment also evaluated the overhead involved in setting up a connection. For UDP the experiment could only be performed for small batch sizes, since UDP is unreliable. Sending a large number of messages over a UDP socket caused occasional packet loss at the receiver due to buffer overflow. This made it impossible to perform the experiment for UDP using large batch sizes. To complete the experiment, all messages must have been received.

The results for the experiment are displayed in Figures 7.3 and 7.4. Figure 7.3 shows the obtained point estimates on a log-based scale, and Figure 7.4 shows the point estimates and their corresponding confidence intervals on a regular scale. (The confidence intervals are generally too small to be distinguishable.) We can see from Figure 7.3 that the rigid connection establishment used by TCP adds a significant overhead for short-lived connections. The implicit connection establishment and reliable service provided by RUPP, on the other hand, adds only minimal overhead compared to the connectionless unreliable service provided by UDP. When five messages are transmitted in each direction, the use of

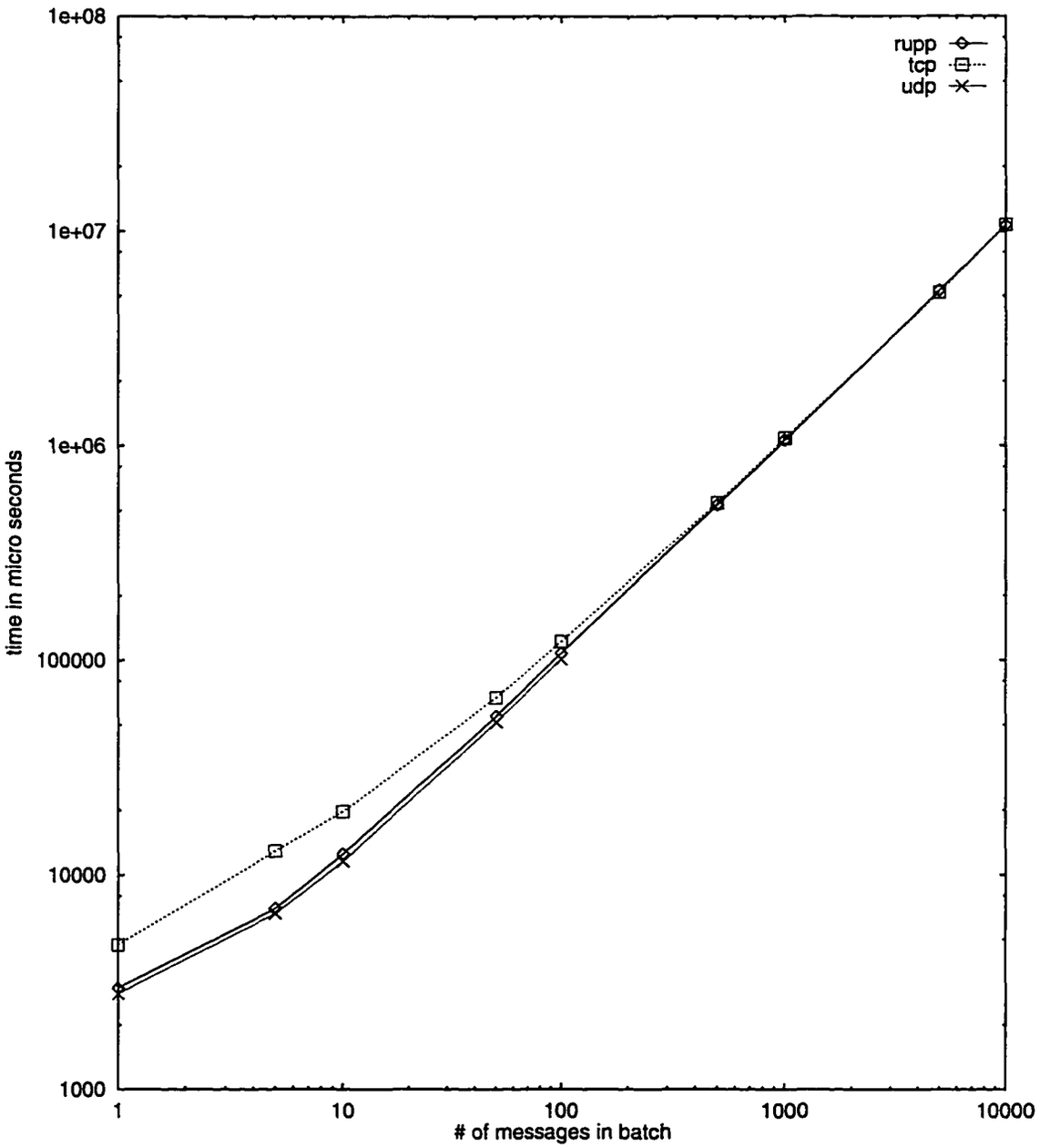

Figure 7.3: Time versus Batch-size

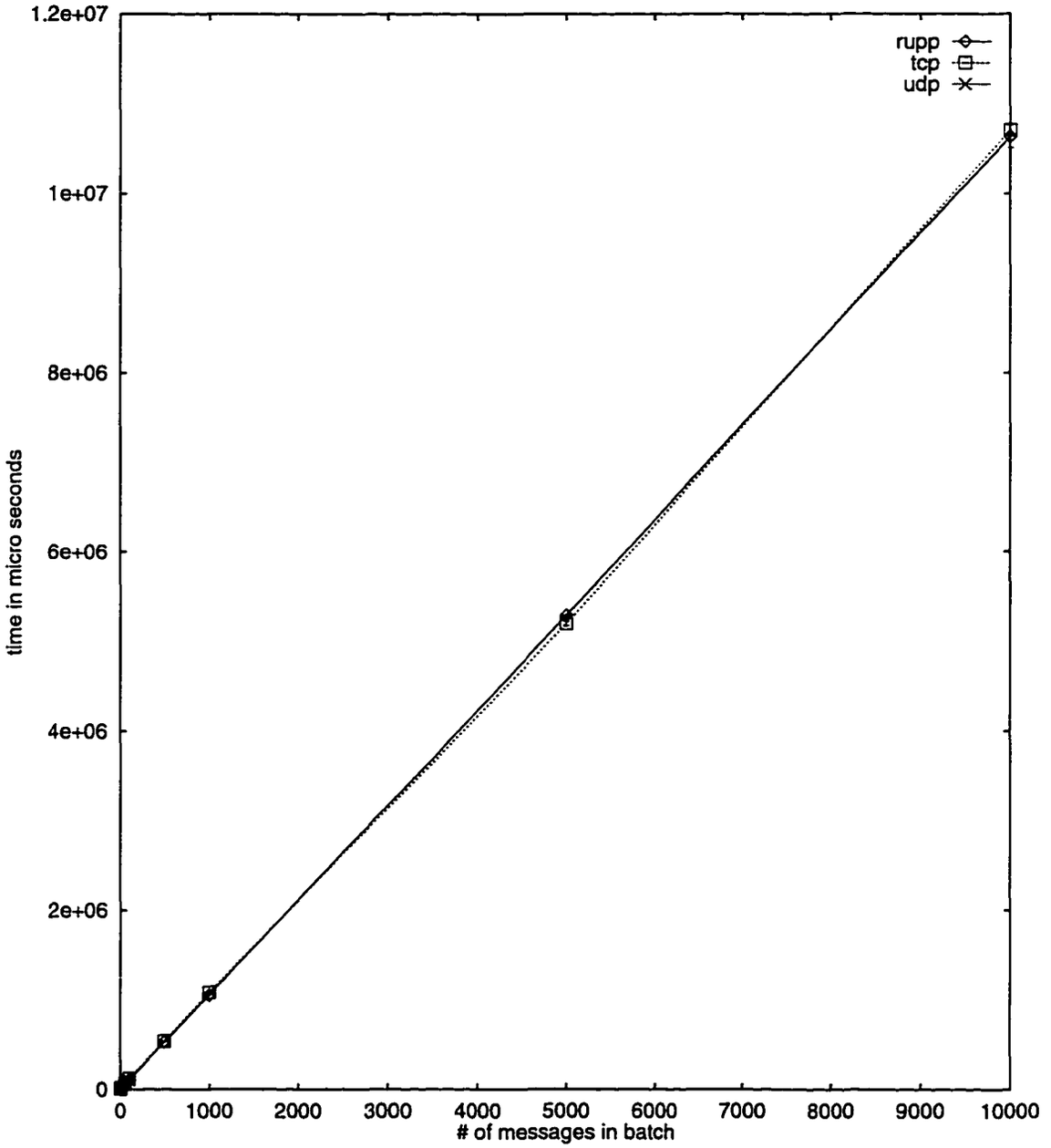

Figure 7.4: Time versus Batch-size

TCP adds approximately 95% overhead compared to UDP. In contrast, the use of RUPP adds only approximately 5% overhead. Our experiment was conducted over a local area network. Over a wide area network, we would expect connection setup for TCP to be even more expensive in comparison to UDP and RUPP.

As the number of messages transmitted increases, the cost for connection establishment becomes less important. For large batch sizes, we see that RUPP and TCP perform similarly. The overhead involved in sending a RUPP segment is less than for TCP, but RUPP is slowed down by occasional buffer overflows in the receiver. The higher overhead of TCP is offset by TCP's more sophisticated flow control mechanism. The net result is comparable performance for the two protocols for bulk data transfer over a non-lossy network. The situation where packets are lost during transmission from one host to another will be considered in a later experiment. As expected, we can see from Figure 7.4 that the time needed to complete the message passing is proportional to the number of messages transmitted.

7.2.2 Request-Response Message Passing (Experiment 2)

In our second experiment we considered the performance of RUPP, TCP, and UDP for request-response type message passing. Instead of transmitting a batch of messages, we transmitted one message, then waited for a reply before transmitting the next message. The pseudo code for the host controlling the experiment is shown in Figure 7.5. As can be seen in Figure 7.5, we still included the cost of connection establishment in our timings and still used messages of size 512 bytes. The number of messages transmitted in this experiment was the same as in the previous experiment but the pattern of communication was completely different. Since there was no risk of buffer overflow for request-response type communication UDP could be included for all numbers of messages.

```
get starting time;
open connection to remote host;
for(i=0; i<num_iteration; i++){
     send 512-byte message;
     receive 512-byte message;
}
close connection;
get ending time;
print out time used;
```

Figure 7.5: Outline of Master Process

The produced point estimates are shown on a log scale in Figure 7.6 and the point estimates and their corresponding confidence intervals are shown on a regular scale in Figure 7.7. Again we can see from Figure 7.6 that the cost of connection establishment is considerable for short-lived connections when using TCP. RUPP, on the other hand, imposes only a small overhead compared to UDP. As before, we can see that the cost of connection establishment is less important when a large number of messages is transmitted. From Figure 7.7 we can see that RUPP outperforms TCP for request-response type communication. As expected, we see that UDP produces the smallest response time. Since UDP supports no reliability, there is less overhead involved in sending a UDP datagram than a RUPP or TCP segment. However, The performance of RUPP stays fairly close to the performance of UDP. The performance difference between UDP and RUPP is less than the performance difference between RUPP and TCP. Consistent with the results from the previous experiment, RUPP adds reliability to UDP at an approximate cost of 5% – a very reasonable overhead for the added functionality of RUPP. For request-response type communication the use of TCP adds an approximate 18% overhead compared to UDP.

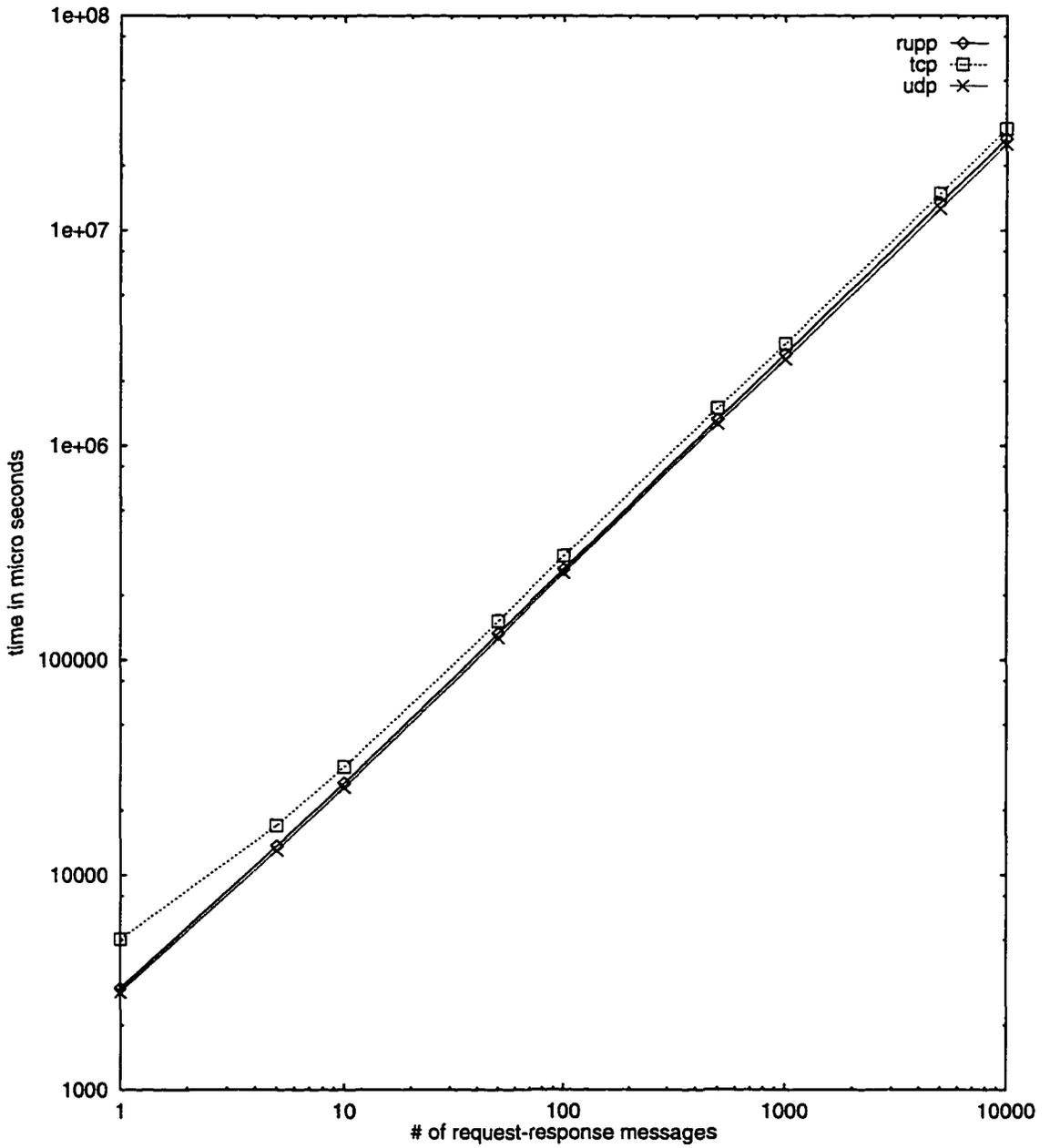

Figure 7.6: Time versus Number of Request-response Messages

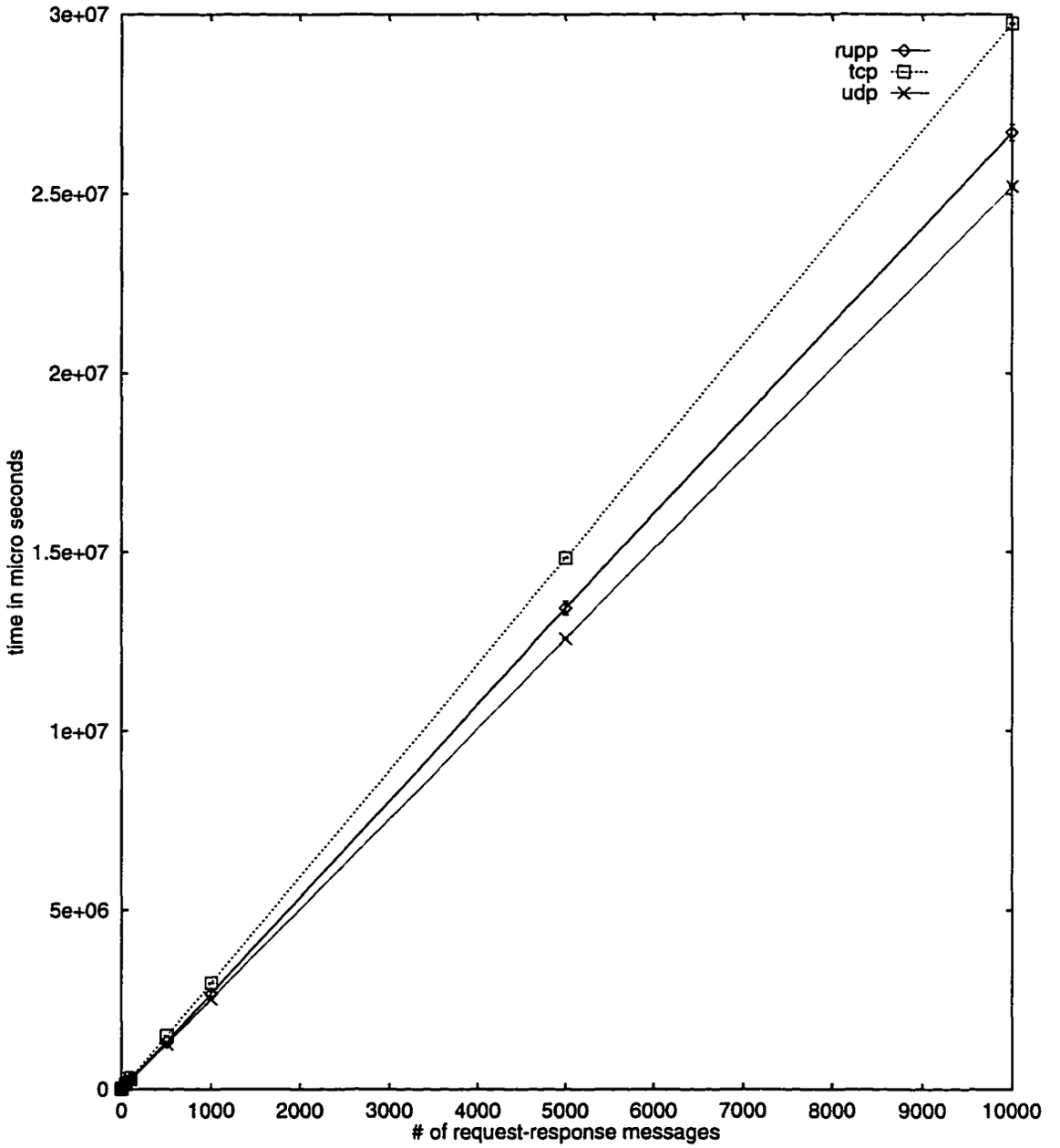

Figure 7.7: Time versus Number of Request-response Messages

7.2.3 Influence of Message Length (Experiment 3)

In our third experiment, we examined the influence of messages size on the performance of our RUPP protocol as well as on performance of the TCP protocol. Experiment 2 was modified to no longer consider connection management and to send a fixed number of request-response messages. Instead of varying the number of messages transmitted we varied the size of the messages. The average round-trip time for a message was calculated and used as one data point in the calculation of our 95% confidence interval (method of batch means). The mean round-trip times and their corresponding confidence intervals are shown in Figure 7.8. Except for the performance of TCP for small messages, the results in Figure 7.8 look as we would expect. The round-trip time needed for a message is proportional to the size of the message. Consistent with Experiment 2, the round-trip time for a message over RUPP is considerably smaller than the round-trip time of a message of equivalent size over TCP. We do not have an explanation for the poor performance of TCP for small messages, but we repeatedly verified the behavior. We are merely happy to observe that our RUPP protocol does not exhibit this strange behavior. The results shown for RUPP in Figure 7.8 serves as validation for our RUPP implementation.

7.2.4 Performance Over a Lossy Network (Experiments 4 and 5)

As mentioned earlier, our experiments were run over an Ethernet without competing traffic. No packets were ever lost or corrupted during transmission from one host to another. However, over wide area networks or wire-less networks, packets can get lost or reordered. The performance of the RUPP protocol is more interesting when packets get lost. Only when packet loss or packet reorder occurs can we expect to see a gain from unordered communication compared to ordered communication. Not until packet loss occurs are we able

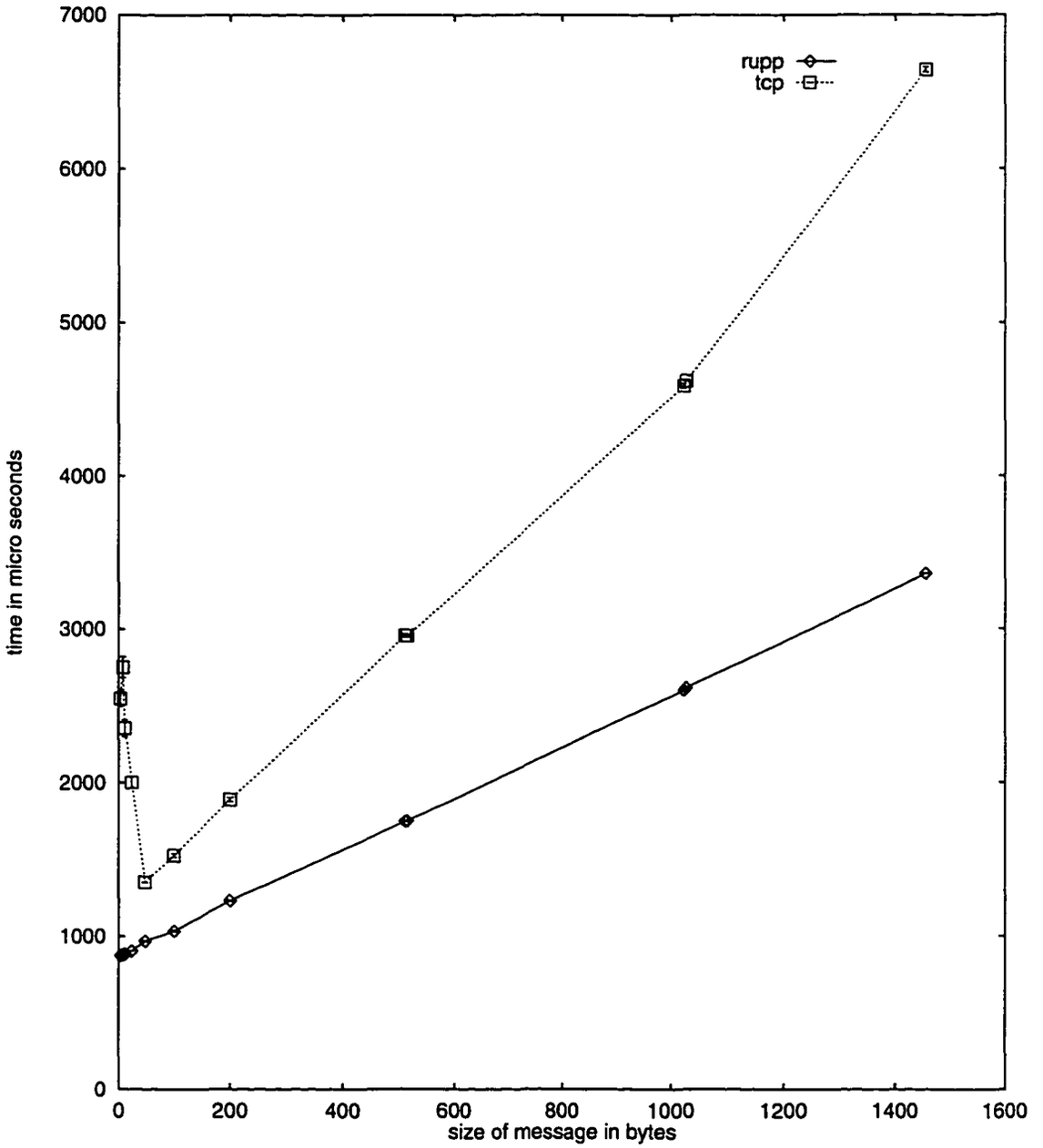

Figure 7.8: Round-trip Time versus Message Size

to evaluate the performance of our reliability mechanisms. As described above, we implemented an option to probabilistically discard arriving packets to evaluate performance in the presence of packet loss.

Performance over a lossy network is most interesting for bulk data transfer. When request-response communication is used, there is no difference between ordered and unordered communication, and the only implementation parameter of real importance is the retransmit timeout value. For bulk data transfer, on the other hand, we can expect unordered communication to offer performance benefits and the full capacity of our reliability mechanisms to be tested. Thus, to test performance over a lossy network, we slightly modified Experiment 1. Connection management was no longer considered in our timings and a fixed number of 1000 messages were transmitted in each batch. The characteristics of the lossy network were varied between the various runs.

We first considered performance when packets were randomly lost; that is, an independent Bernoulli trial was performed for each incoming packet to decide if it should be dropped or not. The results for RUPP and TCP for varying loss probabilities are shown in Figure 7.9. This figure illustrates the great performance benefits of unordered communication. For instance, when the probability of packet loss is 0.02, the message transmission time required by TCP is more than three times greater than the time required by RUPP. This is primarily due to the sequenced delivery enforced by TCP. Thus, we can see that, for applications that require reliability but not order, the use of TCP imposes unnecessarily high overheads. We can see from Figure 7.9, that the slow-down for TCP is roughly proportional to the loss probability. In contrast, RUPP performs very well as long as the probability of packet loss is not too great. For large packet loss probabilities performance of RUPP starts to degrade at a rate similar to the rate of TCP. Although this trend should be noted, we see that RUPP

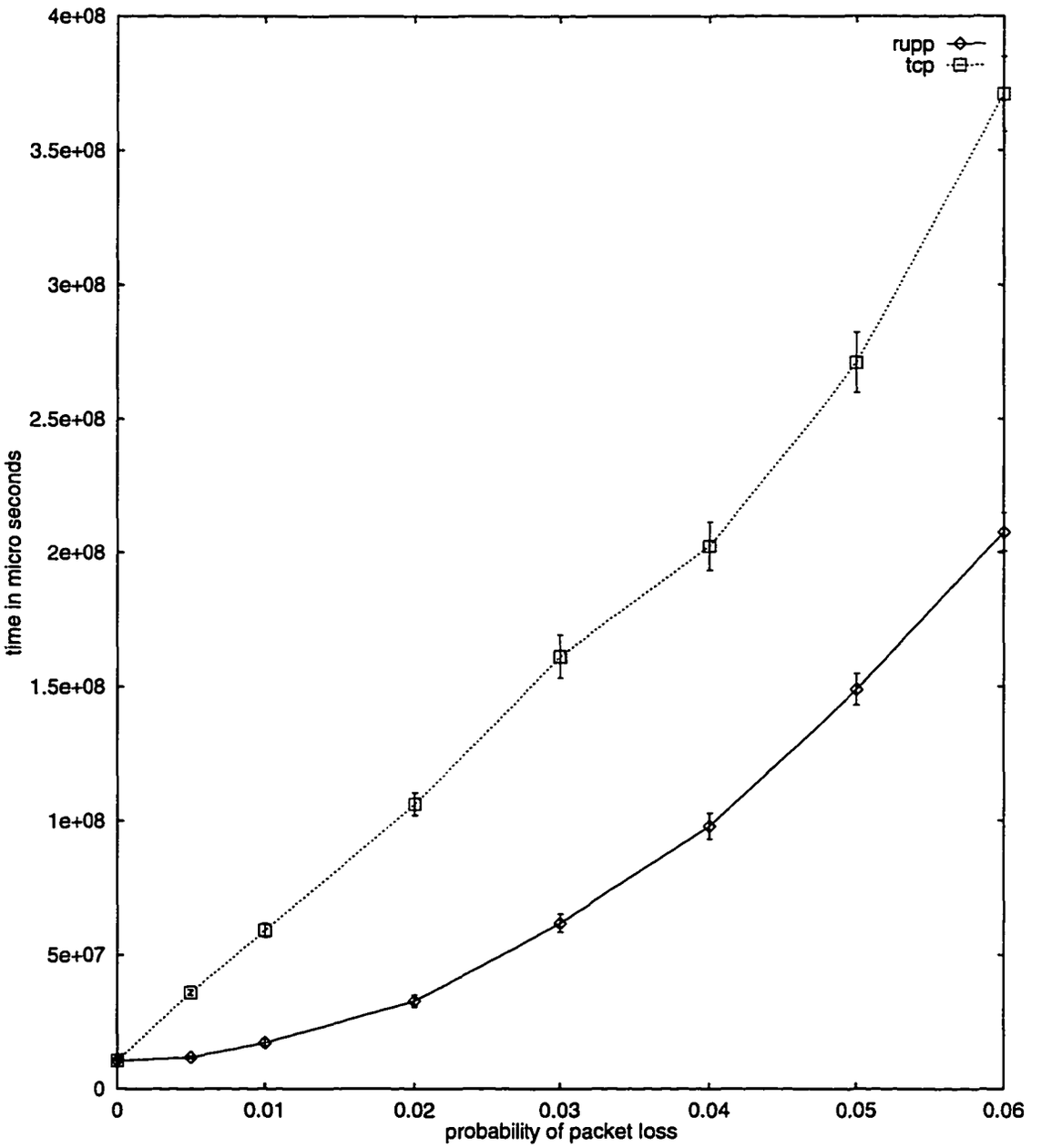

Figure 7.9: Time versus Loss Probability

Probability of entering loss state	Holding time in microseconds	Approximate number of packets lost in loss state
0.016	300	1
0.008	850	2
0.004	1950	4
0.002	4150	8

Table 7.1: Parameters Used in Experiment 5

performs very well for loss probabilities within the range of practical interest. When the probability of packet loss is 0.005, the slowdown for RUPP compared to its performance over an error-free channel is around 12%, whereas the corresponding slowdown for TCP is approximately 235%.

Our next experiment was designed to evaluate the behavior of the protocols for various types of loss behavior. We considered the protocols for varying degrees of bursty error behavior. As described above, we associated a holding time with the "loss" state in our Markov chain to simulate bursty packet loss. The holding times between experiment runs were chosen to allow an increasing number of packets to be lost when the loss state was entered. From Experiment 6, described below, we know that the time to send a 512-byte message is approximately 550 microseconds. Thus, during bulk data transfer, a segment arrives at the receiver approximately every 550 microseconds unless recovery or flow control measures are invoked. Based on this knowledge, appropriate holding times could be selected. The total number of packets lost during an experiment was held approximately constant by adjusting the probability of entering the loss state to compensate for the increased burstiness. The holding times used, the corresponding probabilities, and the approximate number of packets lost each time the loss state was entered are shown in Table 7.1. The

results for RUPP and TCP for the various holding times are shown in Figure 7.10. We can see from Figure 7.10 that the performance of the RUPP protocol is not greatly affected by the burstiness of the packet loss. The performance when packets are lost at random is similar to performance when packets are lost in bursts as long as the overall probability of losing a packet is roughly the same. Since the RUPP protocol allows unordered delivery of messages and retransmits packets selectively, the cost of recovery for a lost packet is roughly the same whether the packets are lost at random or in bursts. The TCP protocol, on the other hand, performs much better when packets are lost in bursts. For TCP the number of times recovery must be performed is more important than the number of packets lost each time recovery is needed. Since TCP provides ordered delivery, each time a segment is lost, TCP must delay the receiver until the segment has been retransmitted. This delay is roughly the same whether a single segment or a sequence of segments are lost. In addition, the Linux implementation of TCP retransmits all segments in the send-queue when a retransmit timeout occurs, incurring a similar cost regardless of the number of segments lost. The cost of congestion avoidance and slow start[41] is also fairly unaffected by the number of segments lost.

7.3 Delivered Experiments (Experiments 6 and 7)

Our second set of experiments considered the performance of the *delivered* and *delivered_all* system calls. The use of the *delivered* and *delivered_all* constructs can reduce network traffic by alleviating the need for user-level acknowledgments. We would also expect the use of *delivered* and *delivered_all* to be more efficient in terms of time compared to user-level acknowledgments. To evaluate the differences we measured the time needed to send and acknowledge a message at the user level compared to the time needed to execute a *send*

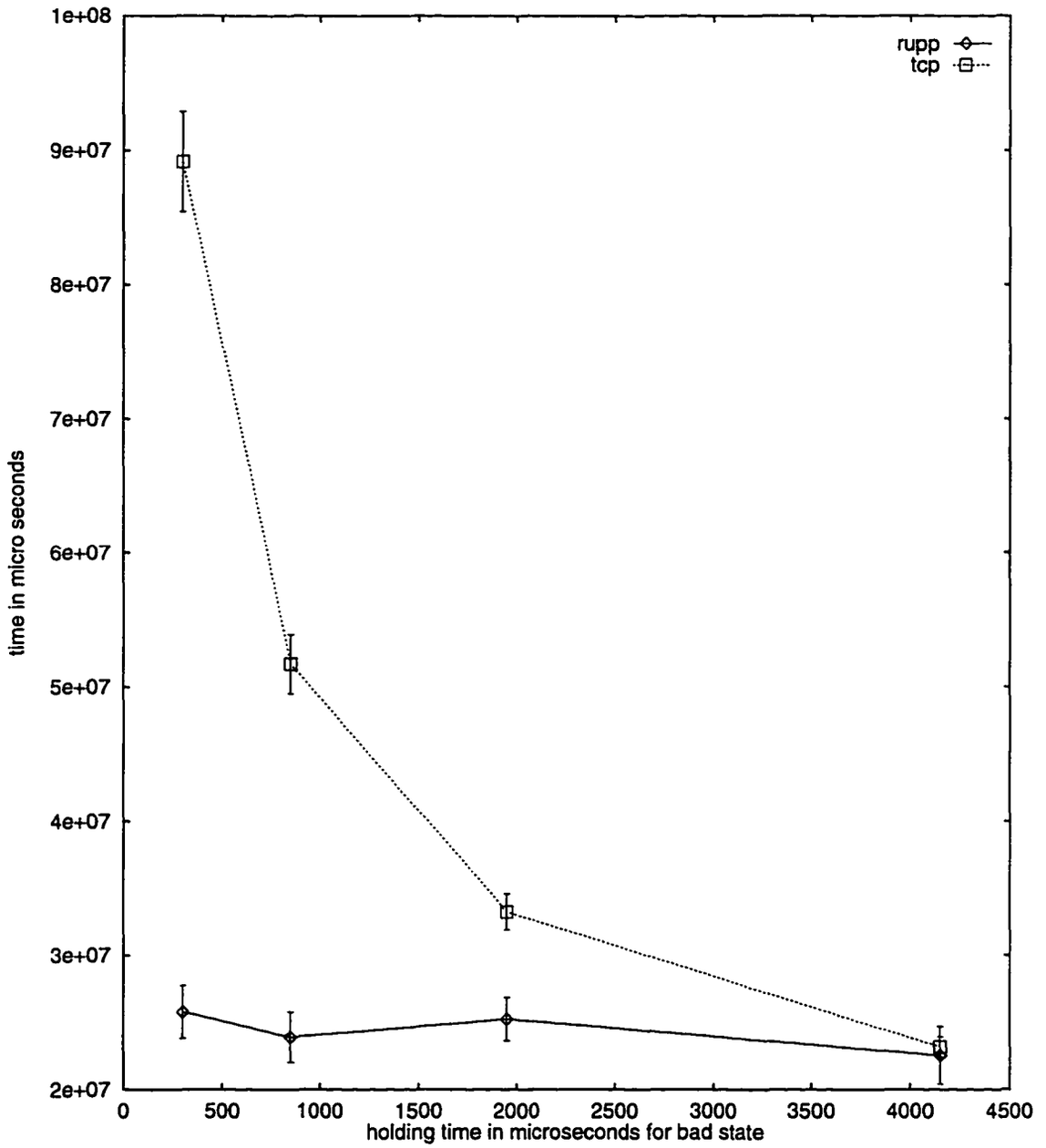

Figure 7.10: Time versus Holding Time in Loss State

directly followed by a *delivered*. For our timings with *delivered*, we enabled the RUPP option that acknowledges messages without delay. We varied the size of the message transmitted, but used a constant size of four bytes (one integer) for the user-level acknowledgment. To evaluate the worst-case overhead of our *delivered* system call, we also measured the time needed to execute a *send* system call. As in Experiment 3, we used the method of batch means to calculate 95% confidence intervals. The results of our experiment are shown in Figure 7.11. We can see that the use of user-level acknowledgments adds a fairly constant overhead compared to using the *delivered* system call. When the size of the transmitted message is small the use of user-level acknowledgments adds an approximate 20% overhead compared to the use of *delivered*. As the size of the message increases, the overhead becomes less significant since the transmission cost of the message is the dominant cost.

Comparing the delay for a *send* operation to the delay for a *send* operation immediately followed by a *delivered* call, we see that, in the worst case, the delay of a *delivered* call is approximately 2.5 times the delay of the send operation. In practice we would expect communication events to be interspersed with processing, and the delay for a *delivered* call would be much smaller. The discussion above focused on our *delivered* system call. However, replacing the call to *delivered* by a call to *delivered_all* produced virtually identical results since the operation was performed after every send operation.

The results displayed in Figure 7.11 also support our analytical results derived in Chapter 3, where we analyzed the message delay for causally ordered communication; we considered the delay from when a process is ready to send a message until this message is available for receipt to the application in the receiving host. We can see from Figure 7.11 that just the send delay for a 1000-byte message is approximately the same as the delay to send and execute a *delivered* for a 100-byte message. For a system with 15 processes

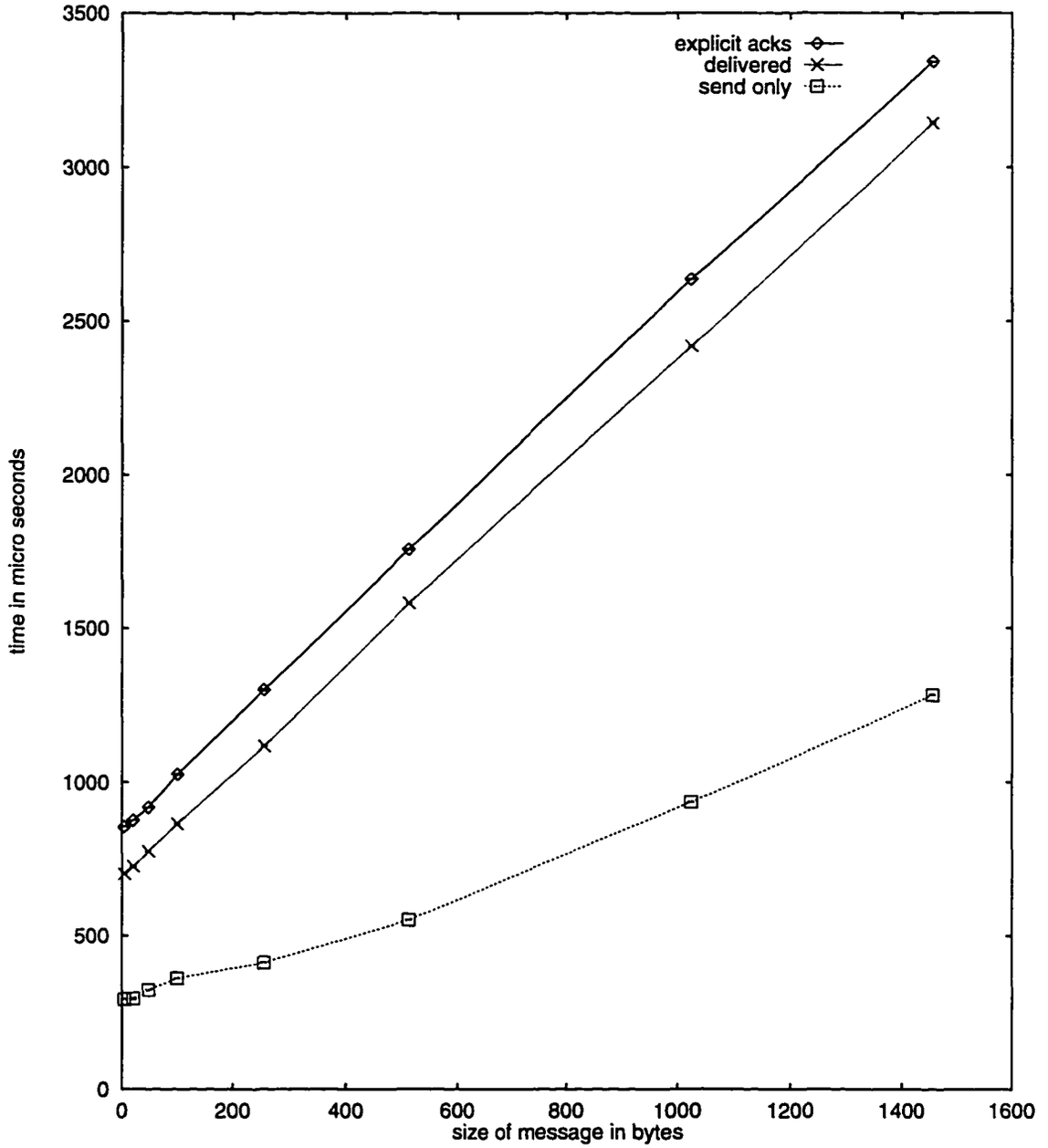

Figure 7.11: Time versus Message Size

and point-to-point communication, the vector time information needed to support causally ordered communication adds 900 bytes to each message. For such a system, the delay to send a 1000-byte message is a lower bound on the message delay for a 100-byte message in a vector time based implementation of causally ordered communication. (Since the cost to acknowledge a message is constant, Figure 7.11 also shows that the actual delay is significantly larger.) The delay to send and execute a *delivered* call for a 100-byte message is an upper bound on the message delay for a 100-byte message when our proposed implementation is used. Thus our experiment suggests that, over a local area network, our implementation of causally ordered communication is more efficient than the vector time implementation for medium or large size systems.

The timings for a *send* operation directly followed by a *delivered* call illustrated the worst-case behavior of our *delivered* construct. Since *delivered* is invoked directly after the *send* operation, we have to wait a maximal amount of time before the acknowledgment is received. Normally, other processing would be interspersed between the *send* call and the *delivered* call for a message and the delay would be smaller. In the best case, the message has already been acknowledged when *delivered* is invoked. The delay for the *delivered* call is then merely the cost of the kernel call that retrieves this information. To evaluate the delay for our *delivered* system call in the best case, we measured the time needed to execute a *delivered* call for a message that was known to be delivered. The best-case delay for the *delivered* call in comparison to the delay for a *send* call is shown in Table 7.2. We can see that in the best case the overhead for the *delivered* call is negligible compared to the delay of sending a message. Even for a very small message (4 bytes), the delay for the *delivered* call is merely a rough 1% of the delay for the *send* operation. As the size of the transmitted message increases the cost of the *delivered* call becomes even less significant. Again, the

Operation	Time in microseconds
delivered	3.30 +/- 0.08
send (4-byte message)	293.12 +/- 0.34
send (512-byte message)	551.34 +/- 0.99

Table 7.2: Best-case Behavior of Delivered

delay for a *delivered_all* call when all messages are already delivered is virtually identical to the delay for *delivered.*

7.4 Flush Channel Experiment (Experiment 8)

Our final experiment was designed to evaluate the performance of our flush channel implementation. We compared the time needed for bulk data transfer using TCP to the time needed to transmit the data as batches of messages over a flush channel. As mentioned in Chapter 4, many multimedia applications require only a partial receipt order on messages, where the messages are naturally partitioned into batches. We considered partitioning the batches of ordinary messages by each of the three flush message types. The experiment where the batches are separated by two-way flush messages is representative of the image transmission example mentioned in Chapter 4. In our experiment we focus on the message passing and do not consider any of the other processing involved. Similar sample applications can be given for the other flush message types[19].

The performance of our flush channel implementation compared to TCP is most interesting for an unreliable network. Only when packets get occasionally lost or reordered can we expect any major performance gains by relaxing the ordering constraints on messages.

We used a network where packets are lost at random with a probability of 0.01. The performance for our flush channel implementation compared to TCP is shown in Figure 7.12 for various batch sizes. Performance for each one of the flush message types is displayed. The performance for TCP is of course unaffected by the batch size and was found in Experiment 4. To give our flush channel implementation a fair chance, we turned on the RUPP option to acknowledge messages without delay for small batch sizes. We did not use the option for large batch sizes. Turning on the option for large batch sizes only hampered performance. The overhead of acknowledging every message in addition to the overhead imposed by the processing for artificial packet loss slowed down the receiver enough to cause frequent buffer overflows. The cost of recovering from buffer overflows was greater than the gain of not having to wait for the acknowledgment timer to expire.

The results displayed in Figure 7.12 are what we would expect. For very small batch sizes, it is more expensive to use our flush channel implementation than to use TCP. When the batch size is very small, substantial synchronization is imposed by our flush channel protocol. As the batch size increases, our flush channel implementation starts to perform more favorably. Even for a batch size of 49, our flush channel implementation clearly outperforms TCP for the considered network. For large batch sizes, the synchronization imposed by our flush channel protocol is negligible compared to the cost of data transfer. Our flush channel implementation offers great performance benefits compared to TCP for large batch sizes. As mentioned in Chapter 4, we expect most applications that use flush channels to transmit large batches of ordinary messages. The results shown in Figure 7.12 verify that flush channel communication can indeed offer performance benefits for such applications.

The performance of the three flush message types in comparison with each other is also

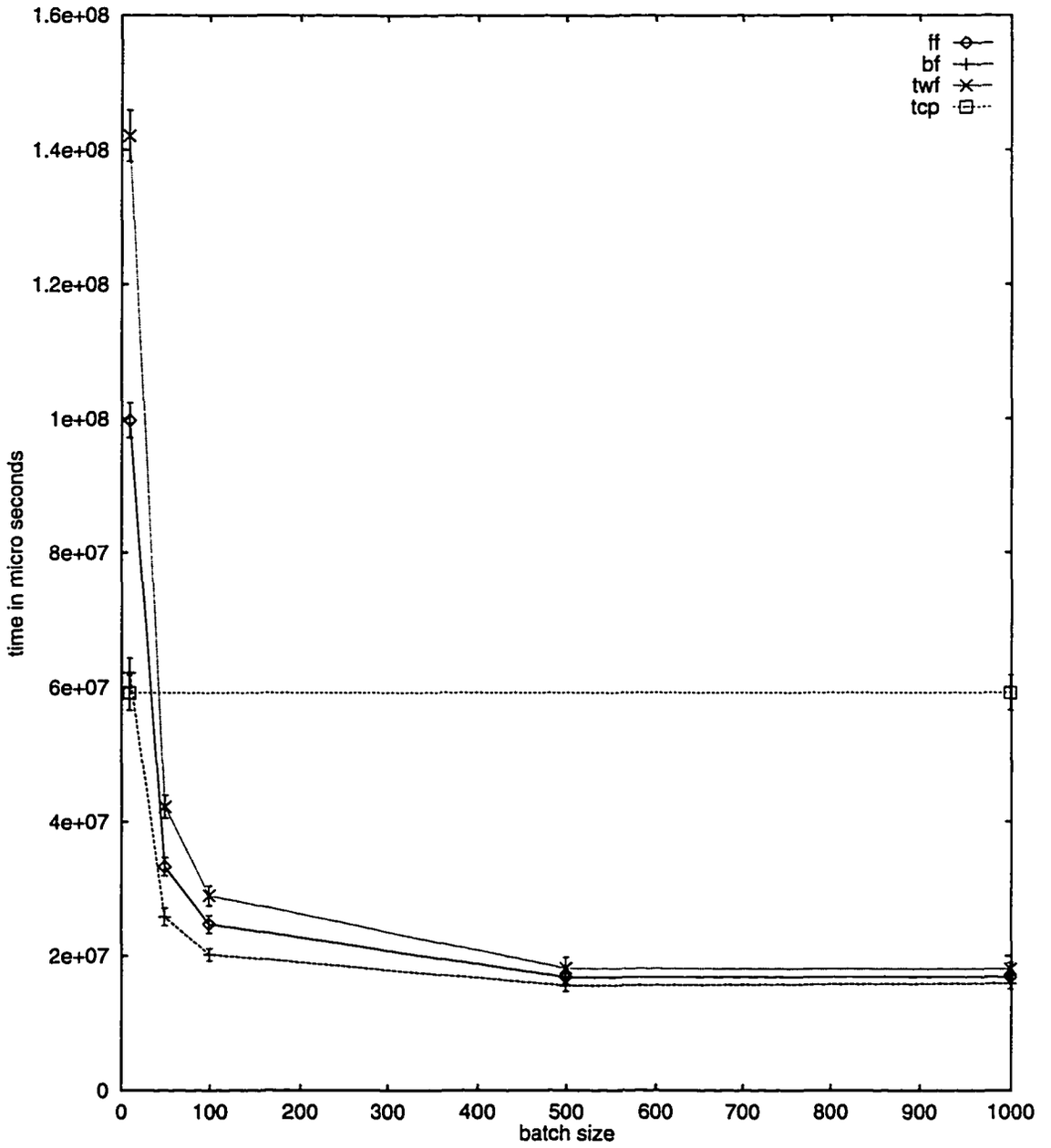

Figure 7.12: Time versus Batch-size

consistent with our expectations. For small batch sizes, we can see that separating the batches by two-way flush messages is the most expensive. The two-way flush messages impose the most synchronization since a *delivered_all* system call is executed before sending a two-way flush message and a *delivered* system call is executed before sending the ordinary message immediately succeeding the two-way flush. The forward flush messages require only a *delivered_all* call and the backward flush messages are the most efficient requiring only a *delivered* call. Again, for large batch sizes, the synchronization imposed by the flush messages are negligible and there is no significant difference between the three flush message types.

The flush channel experiment was carried out over a network with random packet loss. From Experiment 5 we know that TCP performs better when packets are lost in bursts. Hence, for bursty packet loss we would expect the difference in performance between TCP and our flush channel implementation to decrease. The implementation overhead for bursty packet loss is greater than the implementation overhead for random packet loss. Bursty packet loss in combination with the RUPP option to acknowledge messages without delay slows down the receiver enough to prohibit meaningful experiments. Our flush channel experiment therefore focused on a system with random packet loss.

7.5 Summary

This chapter presented some experimental results obtained from our prototype implementation. Our experiments show that RUPP outperforms TCP for short-lived connections and for request-response type communication. Compared to UDP, our RUPP protocol adds reliability at only a small overhead of approximately 5%. For bulk data transfer over long-lived connections and an error-free network, the performance of RUPP and TCP is comparable.

To consider performance over an unreliable network, we implemented a socket option to drop randomly selected incoming packets. For random packet loss, our RUPP protocol offered great performance benefits over TCP. We saw that the number of packets lost is most critical to the performance of RUPP. Performance is not greatly affected by whether the packets are lost at random or in bursts. For TCP, on the other hand, the number of times recovery must be invoked is most critical. TCP performs much better when the packets are lost in bursts since this leads to less recovery invocations. The number of packets lost each time an error condition occurs is less critical.

The performance of the *delivered* and *delivered_all* system calls was also considered in our experiments. We considered both worst-case and best-case performance. In the worst case, when a *delivered* or *delivered_all* immediately succeeds a *send*, a fairly substantial overhead is encountered. However, in practice we expect to have interspersed processing, hence the delay will be much lower. In the best case, when the message (messages) has (have) already been acknowledged before *delivered* (*delivered_all*) is invoked, the overhead is negligible. In practice, the cost of *delivered* and *delivered_all* would also commonly be amortized over multiple send operations. Only for select message would our primitives be invoked. Thus the overhead imposed by our primitives would generally be low.

Our final experiment evaluated the performance of our flush channel implementation over a network with random packet loss. We considered batched data transfer, separating batches of ordinary messages by each one of the three flush message types. As expected, performance improved for larger batch sizes. For medium and large size batches our flush channel implementation clearly outperformed TCP. Our experiments illustrated that relaxed constraints on the message receipt order can indeed offer great performance benefits.

The experimental results, presented in this chapter, validate the design and implemen-

tation of the RUPP protocol. They show the feasibility of our *delivered* and *delivered_all* as well as our flush channel implementation. Our experiments also identify the areas where the RUPP protocol can be improved. The bulk data transfer experiments show that performance of RUPP is hampered by occasional buffer overflows at the receiver. The number of buffer overflows that occur has a direct impact on performance. Efforts to improve the design of the RUPP protocol should be focused on the areas of flow control and buffer management. Since the design and implementation of RUPP is in its infant stages compared to TCP, it is safe to assume that the performance benefits of RUPP displayed by our experiments are due to inherent performance benefits of a reliable unordered transport protocol.

Chapter 8

Concluding Remarks

This chapter contains a summary of the results presented in this dissertation and suggests some directions for further research.

8.1 Dissertation Summary

This dissertation has examined several aspects of the use of transport layer information for distributed systems. It provides support for our thesis that low-level information available at the transport layer may be used to our advantage when solving problems in distributed systems. As was shown in this dissertation, knowledge of the delivery of messages can be a useful tool for distributed systems design.

Both formal and practical aspects of the use of low-level information were considered. A bipartite system model that allows us to reason formally about transport layer information was developed. Based on our model, we developed methods to propagate transport layer information to the user level. We applied this information to design alternative solutions to several well-known problems in distributed systems. We showed that our ideas are not

only theoretically sound and intellectually appealing, but also implementable in practice through a prototype implementation and experimental results.

8.1.1 Formal Developments

We began our development by defining a bipartite system model. Our bipartite system model is needed to allow formal reasoning about transport layer information. The previous system model used for reasoning about distributed system did not allow any distinction between when a message is *delivered* to the transport layer at the receiving host and when it is *received* by the application. Our system model is a straightforward extension of the previous model and it explicitly models the transport layer in the receiver as well as the application layer processes. In this dissertation, we focused on a system employing reliable unordered message passing. The proof rules for communication in the system under our bipartite system model were defined. Based on our system model we formally defined two constructs, *delivered* and *delivered_all* which propagate transport layer information to the user layer. The *delivered* and *delivered_all* constructs allow us to utilize transport layer information at the user level in a sound fashion. The proof rules for our constructs were established. We illustrated with a small example how the proof rules may be employed to show program correctness.

8.1.2 Message Ordering Protocols

We next applied our *delivered* and *delivered_all* constructs, and the transport layer information they provide, to solve some well-known problems in distributed systems. As was illustrated in this dissertation, one major application of our constructs is in the design of message ordering protocols at the user level. As long as messages are passed in FIFO or-

der from the transport layer to the application layer, knowledge of delivery of messages is as powerful as knowledge of receipt of messages in terms of order. As a result, our constructs are extremely useful for providing application-specific ordering constraints. We designed user-level implementations of both causally ordered communication[89] and flush channels[5].

Causally ordered communication can greatly simplify system development but its use is impeded by a high implementation cost. Whether the system should provide causally ordered communication or not has caused heated debate in the distributed systems community. We presented a user-level implementation based on our *delivered_all* construct. Our user-level implementation provides causally ordered communication in a flexible manner. It alleviates the need for building expensive support into the underlying system and inflicts the cost of causal order only on applications which use it. Our performance analysis showed that our suggested implementation performs very well compared to previously suggested implementations for medium and large size systems. As an extension to causally ordered communication, we also considered causal order on a message-by-message basis. Enforcing causal ordering constraints on only a subset of the messages in a computation is often sufficient for correctness and can amply improve performance. We introduced the notion of a causally early message and provided an implementation based on our *delivered_all* construct.

Flush channels are a useful communication primitive for applications such as multimedia which require only a partial receipt order on messages. Previously proposed implementations were designed to build flush channels into the underlying communication subsystem. This is inflexible and imposes an overhead on the system as a whole. Although flush channels can be very useful, they are not a standard construct we can expect the underlying communication subsystem to support. We proposed a user-level implementation of flush

channels based on our *delivered* and *delivered_all* constructs. Our user-level implementation allows applications to benefit from the relaxed ordering constraints and greater efficiency offered by flush channels without requiring kernel-level support for the construct. As shown by our performance analysis, our suggested implementation performs best when the number of flush messages is small compared to the number of ordinary messages.

8.1.3 Transport Layer Vector Time

No common clock exists in a distributed system. Events in the system are only partially ordered based on causality[53]. Vector time was introduced[57, 34] as a means for capturing the causal relationship in the system. It has become a standard tool used in the design of algorithms for distributed systems. Vector time is traditionally perceived with respect to the application layer; it is updated by communication events as they occur at the application layer. Our dissertation considers the impact and use of transport layer information. Considering vector time as perceived at the transport layer is therefore important. Examining the causal relationships present at the transport layer as opposed to the application layer can provide considerable insight on how to better utilize transport layer information.

We established the formal relationships present between transport layer and application layer vector time. Any causal relationships present at the application layer between events in different processes are preserved at the transport layer. The transport layer is also more well informed about the "current time" in the system than the application layer. We showed how this can be very useful for certain types of algorithms. Algorithms that benefit from a reduction of the amount of concurrency in the system can gain in both computational efficiency and accuracy through the use of transport layer vector time. One important class of algorithms that falls into this category is the class of algorithms for distributed debugging.

Debugging a distributed program is an inherently hard problem. Reducing the amount of concurrency in the system is the key to simplifying the debugging process. The use of transport layer vector time also provides the possibility to let acknowledgment messages update vector time. The additional information conveyed by acknowledgment messages can provide us with a more accurate view of the global state of the system. This can further improve efficiency for algorithms that construct a global view of the system. Updating vector time for acknowledgment messages also allows algorithms to utilize the causal relationships introduced by the acknowledgments. We derived a vector time based distributed termination detection algorithm which take advantage of this possibility. Our distributed termination detection algorithm uses causal information propagated through acknowledgments and thus requires vector time to be updated by acknowledgments. Our distributed termination detection algorithm is an excellent example of how low-level information, available virtually for free but traditionally neglected, can be employed to our advantage.

8.1.4 Prototype Implementation

To show practical evidence in support of how transport layer information can be used, we designed and implemented the RUPP transport layer protocol. Support for propagation of information about delivery of messages to the application is an explicit part of the RUPP specification. RUPP does not only offer message delivery information, but is a full-fledged transport layer protocol providing a reliable unordered message passing service to the application. Some of the main components of RUPP include: lightweight connection management; reliable delivery of messages based on sequence numbers, a checksum algorithm, a receive bitvector to remember segments received out of order, and an acknowledgment retransmit algorithm that includes a fast retransmit rule; window-based flow control; and

measures for buffer management. Our prototype implementation was carried out within the Linux operating system. The RUPP protocol was incorporated into the networking code of the Linux kernel (version 1.2.0). The RUPP protocol is accessed through the standard socket interface in Linux. RUPP sockets support all operations that can be performed on a UDP socket. Two new system calls, *delivered* and *delivered_all*, were implemented to obtain information about delivery of messages from a RUPP socket; *delivered* and *delivered_all* are only supported for RUPP sockets. We also implemented a flush channel library on top of RUPP. Two library routines, `flush_send` and `flush_sendto`, were defined. The flush channel library routines take the message type as a parameter in addition to the standard parameters needed by *send* and *sendto*.

We performed several experiments with our RUPP protocol. Our experimental results verified the feasibility of our RUPP design and implementation. The RUPP protocol outperforms TCP for short connections and for "request-response" type communication. The performance of RUPP and TCP is roughly equal for bulk data transfer over an error free network. When packets are lost during transmission from one host to another RUPP can offer performance benefits over TCP, especially when packet loss is fairly random. Our experimental results also showed that the overhead for *delivered* and *delivered_all* are generally low. In practice, communication is interspersed with processing and the cost of *delivered* or *delivered_all* is commonly amortized over multiple send operations. The performance benefit of using transport layer acknowledgments as opposed to explicit user-level acknowledgments was also validated. Additionally, our experimental results illustrated the potential performance benefits of flush channel communication.

8.2 Future Research Directions

The work presented in this dissertation illustrates the great potential for the use of transport layer information. A summary of our research accomplishments was provided in the previous section. In this section we identify several open problems that warrant further investigation.

8.2.1 Traditional Problems

In previous chapters, we used our *delivered* and *delivered_all* constructs to design user-level implementations of causally ordered communication and flush channels. Our distributed termination detection algorithm can also be phrased in terms of our constructs. Many other application areas for our constructs still remain to be investigated. We will explore how our constructs can be used in algorithms for solving several other traditional problems in distributed systems. A few candidate problems are described below.

Distributed Snapshot

Distributed snapshot[20] is a suitable application area for our constructs. A distributed snapshot is a collection of local process states and messages in transit which form a consistent global state. A consistent global state is a global state that could possibly have occurred during the computation. If an event e is recorded in the snapshot, then all events which causally precede e must also be recorded. Taking a meaningful snapshot of a distributed system is a nontrivial task due to the lack of a common time frame and the lack of shared memory. A distributed snapshot can be used for determining stable properties such as distributed termination or for checkpointing. Another application area is the computation of monotonic functions of the global state such as lower bounds on the global virtual time reached by distributed simulations [44, 43]. Several distributed snapshot algorithms have

been proposed[94, 20, 52, 59]. We believe our constructs can be applied to algorithms from the literature in order to make them more efficient. Our constructs can aid a snapshot algorithm in tracking messages in transit.

Optimistic Recovery

Another area where we believe our constructs may be useful is optimistic recovery [99, 93, 45, 32, 107]. Systems employing optimistic recovery take checkpoints and log messages asynchronously. When a process fails, an earlier state of the process is restored by rolling back to the last checkpoint and replaying any messages stored in stable storage. The other processes in the system are rolled back to form a consistent system state. Knowledge of which messages have been successfully delivered should be useful when recovering failed processors and in determining if rollback is necessary for non-failed processors. The problem of optimistic recovery is also interesting from another standpoint. Some recent rollback recovery algorithms are based on vector time[75, 84]. The impact of transport layer vector time on these algorithms should be considered.

Global Predicate Detection

Global predicate detection [27, 95, 35, 38] is a problem closely related to the global snapshot problem. Monitoring the events in a distributed system is difficult due to the concurrency present in the system. Knowledge of when a message is delivered can reduce the amount of concurrency which needs to be considered. We have already seen this indirectly in our work on transport layer vector time. We should also consider if our constructs could be applied directly in the design of global predicate detection algorithms.

8.2.2 Implementation Extensions

The RUPP protocol was designed and implemented as practical evidence in support of how transport layer information could be used. We are considering several extensions to our prototype implementation.

Memory Mapped Send Buffers

As mentioned earlier, much current research effort is spent on providing more efficient implementations of network code. The cost of copying the user data into kernel space accounts for a big portion of the transmission cost for a message. Memory mapped send buffers can be used to alleviate the need for copying the user data into kernel space[72]. Instead of copying the data, the kernel adjusts the memory map to access it. Since both the user code and the kernel are accessing the same memory segments, access to the segments must be synchronized. The user cannot modify the contents of a memory segment until the kernel no longer needs it. Previous implementations control access to the shared segments through semaphores or some similar construct. We believe memory mapped send buffers can be implemented very elegantly for the RUPP protocol. The key observation behind this claim is that our *delivered* construct provides exactly the information needed to control access to the shared memory segments. No explicit synchronization would be needed in the implementation. Instead, to ensure correct transmission of its messages the application would be required to call *delivered* before accessing a segment that may still be used by the kernel. Showing that our *delivered* and *delivered_all* constructs can provide support for memory mapped send buffers would present a strong argument for our constructs. Reducing networking software overhead is the key to allowing high-performance computing applications to be run on a network of workstation.

Causally Ordered Communication

We would like to incorporate our proposed implementation of causally ordered communication into our prototype implementation. Implementing a causally ordered communication library on top of RUPP is not as straightforward as implementing the flush channel library. Flush channels is a connection-oriented construct and could thus be easily implemented on top of a RUPP connection. Causally ordered communication, on the other hand, is a system-wide construct. Implementing it on top of RUPP requires a mechanism for multiplexing between multiple RUPP connections. Another alternative would be to implement a connectionless version of RUPP. Implementing a connectionless reliable protocol is substantially harder, but opens up numerous issues worthy of investigation. We will not elaborate further upon all the issues involved in the design of a connectionless reliable transport protocol. We simply note that this is a topic involving many interesting research problems.

WAN Experiments

To date, all our experiments with RUPP were conducted over a local area network. During normal operation no packets were lost or corrupted during transmission from one host to another. To evaluate performance over an unreliable network, we implemented an option to drop select incoming packets artificially. The packets were dropped at random or in a bursty fashion. We plan to conduct experiments with our protocol between hosts in Sweden and in the US to evaluate its performance over a "true" unreliable network. Comparing these results to the results obtained in our experiments with artificial packet loss will be very interesting. Over a wide area network packets get lost or corrupted for a variety of reasons such as electrical interference, buffer overflow at intermediary gateways, or faulty software along the path. Thus, realistic modeling of packet loss for a wide area network is

hard. Although much theoretical work on the probability of packet loss can be found in the literature, primarily based on queuing theoretic models, experimental results in this area are scarce. The uncontrollable nature of experiments over a wide area network also poses practical problems. A large number of runs will be required to obtain useful statistics.

8.2.3 Alternative Semantics for Delivered and Delivered_all

By definition, a call to *delivered* for a message m will return when message m has been safely delivered to the transport layer at the receiving host. A call to *delivered_all* returns when all messages sent prior to invoking *delivered_all* have been delivered to the transport layer in the receiving host. Our definitions of *delivered* and *delivered_all* are intuitive and easy to understand. However it is also possible to imagine other definitions:

- Our constructs could be implemented as booleans. A call to *delivered* would return true if the message was known to be delivered, false otherwise. Similarly, a call to *delivered_all* would return true if all messages were known to be delivered, false otherwise.

- An interrupt approach is another possibility. A call to *delivered* or *delivered_all* could indicate that an interrupt should be generated once the message or all messages were known to be delivered. The interrupt would put the process performing the call in a specified interrupt handler. The user should be allowed to specify sections of the code where the interrupt activity is temporarily disabled.

- We could associate a time-out value with the constructs. The time-out value would allow the user to specify the maximum amount of time a *delivered* or *delivered_all* call would wait for the delivery of a message. This is similar to the SO_LINGER socket

option[98]. The return value from *delivered* and *delivered_all* would have to indicate whether the call returned due to the time-out expiring or due to successful delivery of the message(s).

- As an extension to the time-out approach, *delivered* and *delivered_all* could be designed in a fashion that allowed them to be part of a modified version of the select system call [97]. Besides examining I/O descriptors for read and write activity, the select call would check if an indicated message had been delivered.

Considering the formal implications of various possible definitions of *delivered* and *delivered_all* is interesting. For instance, a boolean implementation would not allow anything to be concluded when the return value is false, since the message may indeed have been delivered even though the invocation of *delivered* returned false. Most interesting from a formal standpoint may be the interrupt approach. Including the interrupt capability with its associated disabled code segments into our formal framework is a challenging problem.

8.2.4 Delivered and Delivered_all for Broadcast Communication

Our definitions of *delivered* and *delivered_all* refers to point-to-point communication. Considering *delivered* and *delivered_all* for a broadcast context [21, 91, 63] presents another interesting extension. It seems straightforward to generalize our constructs to a broadcast system. We also believe many of the applications suggested for our constructs could be directly extended to a broadcast environment. However, less information is often required to enforce message orderings in a broadcast context. This could make implementations of message ordering protocols that are based on synchronization less competitive.

8.2.5 Formal Verification

We have established the proof rules for our *delivered* and *delivered_all* constructs. As for the proof rules of other communication primitives, the proof rules do not account for the possibility that the constructs could return an error code. In the formal framework, communication commands succeed or hang indefinitely. However, in practice the use of a reliable transport protocol does not guarantee that a message is successfully delivered. The transport protocol will make a fixed number of attempts at delivering the message, after which it gives up and reports an error to the application. We would like to extend the proof rules for communication events to handle error returns. This would be an important step towards narrowing the gap between formal verification and program software.

Similarly, in the formal framework, a *delivered* call on an unsent message is assumed to hang indefinitely. In practice, however, there is no distinction between a message that was never sent and a message that has already been successfully delivered. A *delivered* call for an unsent message would return immediately. Our proof rules could be modified to allow *delivered* calls only for messages that have been sent.

Our developed proof rules can be used to show the partial correctness (safety) of a distributed program. A partial correctness proof ensures that the program behaves correctly if it makes progress[55]. It does not ensure freedom from deadlock. Freedom from deadlock is ensured through a liveness proof. Any impact of transport layer information on the liveness of a distributed program should also be considered.

8.3 Chapter Summary

Transport layer information is traditionally ignored in most work on distributed systems. We believe that transport layer information can be a valuable tool for solving problems in distributed systems. Knowledge of when a message is delivered provides useful information that can be applied to distributed systems design. As demonstrated in this dissertation we have already obtained many promising results in this area. A summary of our research contributions was given in this chapter. Finally we outlined several open problems which we plan to investigate further.

Bibliography

[1] Mark Abbott and Larry Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[2] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, July 1986.

[3] Frank Adelstein and Mukesh Singhal. Real-time causal message ordering in multimedia systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 36–43, 1995.

[4] Subhash Agrawal and Ravi Ramaswamy. Analysis of the resequencing delay for M/M/m systems. *Proc. 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 27–35, 1987.

[5] Mohan Ahuja. Flush primitives for asynchronous distributed systems. *Information Processing Letters*, 34:5–12, 1990.

[6] Mohan Ahuja. An implementation of f-channels, a preferable alternative to fifo channels. *Proc. 11th Int. Conf. Distributed Computing Systems*, pages 180–187, 1991.

[7] Mohan Ahuja, Kannan Varadhan, and Amitabh Sinha. Flush message passing in communicating sequential processes. In S. Navathe N. Rishe and D. Tal, editors, *Parallel Architecture*, pages 31–47. IEEE Computer Society Press, 1991.

[8] Paul D. Amer, Christophe Chassot, Thomas J. Connoly, Michel Diaz, and Philp Conrad. Partial-order transport service for multimedia and other applications. *IEEE/ACM Transactions on Networking*, 2(5):440–455, October 1994.

[9] Twan Basten. Breakpoints and time in distributed computations. In Gerard Tel and Paul Vitanyi, editors, *Proceedings of 8th International Workshop on Distributed Algorithms (LCNS 857)*, pages 340–354. Springer-Verlang, 1994.

[10] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[11] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. Spin – an extensible microkernel for application-specific operating system services machine. *Operating Systems Review*, 29(1):74–77, January 1995.

[12] Kenneth Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, January 1994.

[13] Kenneth Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5:47–76, 1987.

[14] Kenneth Birman, Andre' Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, aug 1991.

[15] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[16] R. Braden, Editor. Requirements for internet hosts – communication layers. RFC 1122, October 1989.

[17] Jerszy Brzezinski, Jean-Michel Helary, and Michel Raynal. Distributed termination detection: General model and algorithms. Technical Report 1964, Unite de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex, France, March 1993.

[18] Tracy Camp. *Flush Communication Channels: Effective Implementation and Verification.* PhD thesis, The College of William & Mary in Virginia, 1993.

[19] Tracy Camp, Phil Kearns, and Mohan Ahuja. Proof rules for flush channels. *IEEE Transactions on Software Engineering*, 19(4):366–378, 1993.

[20] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[21] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[22] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.

[23] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. Technical Report LITP 92.77, Institut Blaise Pascal, Universite Paris 7, 1992. Updated version (Feb. 94) of the tech report received from Mattern.

[24] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. *Opearting Systems Review*, 29(1):83–86, January 1995.

[25] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. *Operating Systems Review*, 27(5):44–57, December 1993.

[26] Robert Cooper. Experience with causally and totally ordered communication support: A cautionary tale. *Operating Systems Review*, 28(1):28–31, January 1994.

[27] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[28] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner: Architectural support for high performance protocols. *IEEE Network*, 7(4):35–43, July 1993.

[29] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[30] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[31] Dennis Edwards and Phil Kearns. DTVS: A distributed trace visualization system. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 281–288. IEEE Computer Society Press, 1994.

[32] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[33] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The operating system kernel as a secure programmable machine. *Operating Systems Review*, 29(1):78–82, January 1995.

[34] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[35] Eddy Fromentin and Michel Raynal. Inevitable global states: a concept to detect unstable properties of distributed computations in an observer independent way. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 242–248. IEEE Computer Society Press, 1994.

[36] Eddy Fromentin and Michel Raynal. Characterizing and detecting the set of global states seen by all observers of a distributed computation. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 431–438, 1995.

[37] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*, chapter 14: Firewall Machines. O'Reilly & Associates, 1994.

[38] Vijay Garg and Craig Chase. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 423–430, 1995.

[39] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[40] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[41] Van Jacobson. Congestion avoidance and control. In *Proceedings ACM SIGCOMM 1988*, pages 314–329. ACM, August 1988.

[42] Van Jacobson. Modified TCP congestion avoidance algorithm. end2end-interst mailing list, April 1990.

[43] D. Jefferson. Virtual time II: Storage management in conservative and optimistic systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–90. ACM Press , New York, NY, USA, 1990.

[44] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[45] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[46] Michael K. Johnson. *Linux Kernel Hackers' Guide*, chapter 2: The /proc filesystem. http://sunsite.unc.edu/mdw/LDP/khg/khg.html.

[47] Laveen N. Kanal and A. R. K. Sastry. Models for channels with memory and their application to error control. *Proceedings of the IEEE*, 66(7):724–744, July 1978.

[48] Phil Kearns, Tracy Camp, and Mohan Ahuja. An implementation of flush channels based on a verification methodology. *Proc. 12th Int. Conf. Distributed Computing Systems*, pages 336–343, 1992.

[49] Phil Kearns and Brahma Koodalattupuram. Immediate ordered service in distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 611–618, 1989.

[50] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, Inc., 1975.

[51] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Lazy replication: Exploiting the semantics of distributed services. Technical Report MIT/LCS/TR-84, MIT Laboratory for Computer Science, 1990.

[52] T. Lai and T. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, 1987.

[53] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[54] Lampson. Hints for computer system design. In *Proc. 9th ACM Symposium on Operating System Principles*, pages 33–48. October 1983.

[55] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[56] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 244–255. ACM Press, December 1993.

[57] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[58] Friedemann Mattern. Distributed termination detection with sticky state indicators. Technical Report 200/90, Department of Computer Science, University of Kaiserslautern, Germany, 1990.

[59] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, August 1993.

[60] Friedmann Mattern. New algorithms for distributed termination detection in asynchronous message passing systems. Technical Report 42/85, University of Kaiserslautern, September 1985.

[61] Friedmann Mattern and Stefan Fünfrocken. A non-blocking lightweight implementation of causal order message delivery. Technical Report TR-VS-95-01, Department of Computer Science, Technical University of Darmstadt, March 1995.

[62] Sigurd Meldal, Sriram Sankar, and James Vera. Exploiting locality in maintaining potential causality. In *Proceedings of the 10th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 231–239, 1991.

[63] P. M. Melliar-Smith, Louise M. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[64] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Implementing fault-tolerant replicated objects using psync. In *Proc 8th Symp. on Reliable Distributed Systems*, pages 42–52. IEEE Computer Society, 1989.

[65] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 39–51. ACM, November 1987.

[66] Sape J. Mullender. *The Amoeba distributed operating system : selected papers, 1984-1987.* Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands, 1987.

[67] Marco Oey, Koen Langendoen, and Henri Bal. Comparing kernel-space and user-space communication protocols on amoeba. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 238–245, 1995.

[68] Shawn Ostermann. *Reliable Message Transport for Network Communication.* PhD thesis, Purdue University, May 1994.

[69] Steve Park. Lecture notes on simulation, 1992.

[70] Steve Park and Keith Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[71] C. Partridge and R. Hinden. Version 2 of the reliable data protocol (RDP). RFC 1151, April 1990.

[72] Craig Partridge and Stephen Pink. A faster udp. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[73] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[74] Sandy Peterson. *Causal Synchrony in the Design of Distributed Protocols.* PhD thesis, College of William & Mary, 1994.

[75] Sandy L. Peterson and Phil Kearns. Rollback based on vector time. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 68–77. IEEE Computer Society Press, 1993.

[76] Jon Postel, Editor. Internet official protocol standards. RFC 1600, March 1994.

[77] Jon Postel. User datagram protocol. RFC 768, August 1980.

[78] Jon Postel, Editor. Internet protocol. RFC 791, September 1981.

[79] Jon Postel, Editor. Internet control message protocol. RFC 792, September 1981.

[80] Jon Postel. Transmission control protocol. RFC 793, September 1981.

[81] Jon Postel. A standard for the transmission of IP datagrams over Ethernet networks. RFC 894, April 1984.

[82] S. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, 1983.

[83] Michel Raynal, Andre' Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39:343–350, September 1991.

[84] Golden G. Richard III and Mukesh Singhal. Complete process recovery in distributed systems using vector time. Technical Report OSU-CISRC-7/94-TR39, Ohio State University, July 1994.

[85] Luis Rodrigues and Paulo Verissimo. How to avoid the cost of causal communication in large-scale systems. In *Proceedings of the 6th SIGOPS European Workshop*, September 1994.

[86] Luis Rodrigues and Paulo Verissimo. Causal separators for large-scale multicast communication. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 83–91, 1995.

[87] Frederic Ruget. Cheaper matrix clocks. In Gerard Tel and Paul Vitanyi, editors, *Proceedings of 8th International Workshop on Distributed Algorithms (LCNS 857)*, pages 355–369. Springer-Verlang, 1994.

[88] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transaction on Computer Systems*, 2(4):277–288, November 1984.

[89] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proc 3rd Internat. Workshop on Distributed Algorithms, Nice, Lecture Notes in Computer Science 392*, pages 219–232. Springer, Berlin, 1989.

[90] R.D. Schlichting and F.B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, 1984.

[91] Fred B. Schneider, David Gries, and Richard D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4:1–15, 1984. (North Holland).

[92] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[93] A. Sistla and J. Welch. Efficient distributed recovery using message logging. In *Procedings of the ACM Symposium on Principles of Distributed Computing*, pages 223–238. 1989.

[94] M. Spezialetti and J.P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[95] M. Spezialetti and J.P. Kearns. Simultaneous regions: A framework for the consistent monitoring of distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 61–69. IEEE Computer Society , Washington, DC, 1989.

[96] Pat Stephenson. *Fast Causal Multicast.* PhD thesis, Cornell University, February 1991.

[97] W. Richard Stevens. *UNIX Network Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1990.

[98] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Company, 1994.

[99] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[100] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[101] G. Tel and F. Mattern. The derivation of distributed termination detection from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, 1993.

[102] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[103] Rodney W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18:33–36, 1984.

[104] Robert van Renesse. Causal controversy at le Mount St. Michel. *Operating Systems Review*, 27(2), April 1993.

[105] Robert van Renesse. Why bother with CATOCS? *Operating Systems Review*, 28(1):22–27, January 1994.

[106] David Velten, Robert Hinden, and Jack Sax. Reliable data protocol. RFC 908, July 1984.

[107] S. Venkatesan. Optimistic crash recovery without rolling back non-faulty processors. Technical Report IS-92-813-A, University of Texas at Dallas, 1992.

[108] Matt Welsh and Lar Kaufman. *Running Linux.* O'Reilly & Associates, first edition, February 1995.

VITA

Anna Karin Brunstrom was born in Karlstad, Sweden, May 14, 1967. She graduated from Sundsta Gymnasiet, Karlstad, with a concentration in Natural Science, June 1986. She attended Pepperdine University, 1987-91, where she received a B.S. degree in Computer Science and Mathematics. In August 1991, the author entered the College of William and Mary as a graduate assistant in the Department of Computer Science. She received a M.S degree in Computer Science in May 1993. The author remained with the Department entering the Ph.D program in August 1993.